Finite difference methods for vibration problems

Hans Petter Langtangen[1,2]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo

Nov 12, 2014

Note: **PRELIMINARY VERSION** (expect typos)

Contents

Vibration problems lead to differential equations with solutions that in time, typically in a damped or undamped sinusoidal fashion. Such solut certain demands on the numerical methods compared to other phenomen solutions are monotone. Both the frequency and amplitude of the osci need to be accurately handled by the numerical schemes. Most of the re and specific building blocks introduced in the fortcoming text can be to construct sound methods for partial differential equations of wave n multiple spatial dimensions.

1 Finite difference discretization

Much of the numerical challenges with computing oscillatory solutions i and PDEs can be captured by the very simple ODE $u'' + u = 0$ a

therefore the starting point for method development, implementation, and nalysis.

.1 A basic model for vibrations

system that vibrates without damping and external forcing can be described y ODE problem

$$u'' + \omega^2 u = 0, \quad u(0) = I,\ u'(0) = 0,\ t \in (0, T]\,. \tag{1}$$

ere, ω and I are given constants. The exact solution of (1) is

$$u(t) = I\cos(\omega t)\,. \tag{2}$$

hat is, u oscillates with constant amplitude I and angular frequency ω. The orresponding period of oscillations (i.e., the time between two neighboring eaks in the cosine function) is $P = 2\pi/\omega$. The number of periods per second $f = \omega/(2\pi)$ and measured in the unit Hz. Both f and ω are referred to as equency, but ω may be more precisely named angular frequency, measured in ad/s.

In vibrating mechanical systems modeled by (1), $u(t)$ very often represents a osition or a displacement of a particular point in the system. The derivative $u'(t)$ nen has the interpretation of the point's velocity, and $u''(t)$ is the associated cceleration. The model (1) is not only applicable to vibrating mechanical ystems, but also to oscillations in electrical circuits.

.2 A centered finite difference scheme

o formulate a finite difference method for the model problem (1) we follow the our steps from Section **??**in [1].

tep 1: Discretizing the domain. The domain is discretized by introducing uniformly partitioned time mesh in the present problem. The points in the nesh are hence $t_n = n\Delta t$, $n = 0, 1, \ldots, N_t$, where $\Delta t = T/N_t$ is the constant ngth of the time steps. We introduce a mesh function u^n for $n = 0, 1, \ldots, N_t$, hich approximates the exact solution at the mesh points. The mesh function ill be computed from algebraic equations derived from the differential equation roblem.

tep 2: Fulfilling the equation at discrete time points. The ODE is to e satisfied at each mesh point:

$$u''(t_n) + \omega^2 u(t_n) = 0, \quad n = 1, \ldots, N_t\,. \tag{3}$$

Step 3: Replacing derivatives by finite differences. The de $u''(t_n)$ is to be replaced by a finite difference approximation. A common order accurate approximation to the second-order derivative is

$$u''(t_n) \approx \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2}\,.$$

Inserting (4) in (3) yields

$$\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} = -\omega^2 u^n\,.$$

We also need to replace the derivative in the initial condition by difference. Here we choose a centered difference, whose accuracy is sir the centered difference we used for u'':

$$\frac{u^1 - u^{-1}}{2\Delta t} = 0\,.$$

Step 4: Formulating a recursive algorithm. To formulate the cc tional algorithm, we assume that we have already computed u^{n-1} and that u^{n+1} is the unknown value, which we can readily solve for:

$$u^{n+1} = 2u^n - u^{n-1} - \Delta t^2\omega^2 u^n\,.$$

The computational algorithm is simply to apply (7) successively f $1, 2, \ldots, N_t - 1$. This numerical scheme sometimes goes under the name St method or Verlet integration[1].

Computing the first step. We observe that (7) cannot be used fo since the computation of u^1 then involves the undefined value u^{-1} at t The discretization of the initial condition then come to rescue: (6) $u^{-1} = u^1$ and this relation can be combined with (7) for $n = 1$ to yield for u^1:

$$u^1 = 2u^0 - u^1 - \Delta t^2\omega^2 u^0,$$

which reduces to

$$u^1 = u^0 - \frac{1}{2}\Delta t^2\omega^2 u^0\,.$$

Exercise 5 asks you to perform an alternative derivation and also to ge the initial condition to $u'(0) = V \neq 0$.

[1] `http://en.wikipedia.org/wiki/Verlet_integration`

The computational algorithm. The steps for solving (1) becomes

1. $u^0 = I$
2. compute u^1 from (8)
3. for $n = 1, 2, \ldots, N_t - 1$:
   (a) compute u^{n+1} from (7)

he algorithm is more precisely expressed directly in Python:

```
= linspace(0, T, Nt+1)  # mesh points in time
dt = t[1] - t[0]        # constant time step
= zeros(Nt+1)           # solution

[0] = I
[1] = u[0] - 0.5*dt**2*w**2*u[0]
or n in range(1, Nt):
    u[n+1] = 2*u[n] - u[n-1] - dt**2*w**2*u[n]
```

Remark.

In the code, we use `w` as the symbol for ω. The reason is that this author prefers `w` for readability and comparison with the mathematical ω instead of the full word `omega` as variable name.

Operator notation. We may write the scheme using the compact difference notation (see Section **??**in [1]). The difference (4) has the operator notation $[D_t D_t u]^n$ such that we can write:

$$[D_t D_t u + \omega^2 u = 0]^n \,. \tag{9}$$

Note that $[D_t D_t u]^n$ means applying a central difference with step $\Delta t/2$ twice:

$$[D_t(D_t u)]^n = \frac{[D_t u]^{n+\frac{1}{2}} - [D_t u]^{n-\frac{1}{2}}}{\Delta t}$$

which is written out as

$$\frac{1}{\Delta t}\left(\frac{u^{n+1} - u^n}{\Delta t} - \frac{u^n - u^{n-1}}{\Delta t}\right) = \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} \,.$$

The discretization of initial conditions can in the operator notation be expressed as

$$[u = I]^0, \quad [D_{2t} u = 0]^0, \tag{10}$$

where the operator $[D_{2t} u]^n$ is defined as

$$[D_{2t} u]^n = \frac{u^{n+1} - u^{n-1}}{2\Delta t} \,. \tag{11}$$

2 Implementation

2.1 Making a solver function

The algorithm from the previous section is readily translated to a co Python function for computing (returning) $u^0, u^1, \ldots, u^{N_t}$ and $t_0, t_1,$ given the input I, ω, Δt, and T:

```
from numpy import *
from matplotlib.pyplot import *
from vib_empirical_analysis import minmax, periods, amplitudes

def solver(I, w, dt, T):
    """
    Solve u'' + w**2*u = 0 for t in (0,T], u(0)=I and u'(0)=0,
    by a central finite difference method with time step dt.
    """
    dt = float(dt)
    Nt = int(round(T/dt))
    u = zeros(Nt+1)
    t = linspace(0, Nt*dt, Nt+1)

    u[0] = I
    u[1] = u[0] - 0.5*dt**2*w**2*u[0]
    for n in range(1, Nt):
        u[n+1] = 2*u[n] - u[n-1] - dt**2*w**2*u[n]
    return u, t
```

A function for plotting the numerical and the exact solution is also co to have:

```
def u_exact(t, I, w):
    return I*cos(w*t)

def visualize(u, t, I, w):
    plot(t, u, 'r--o')
    t_fine = linspace(0, t[-1], 1001)  # very fine mesh for u_e
    u_e = u_exact(t_fine, I, w)
    hold('on')
    plot(t_fine, u_e, 'b-')
    legend(['numerical', 'exact'], loc='upper left')
    xlabel('t')
    ylabel('u')
    dt = t[1] - t[0]
    title('dt=%g' % dt)
    umin = 1.2*u.min();  umax = -umin
    axis([t[0], t[-1], umin, umax])
    savefig('vib1.png')
    savefig('vib1.pdf')
```

A corresponding main program calling these functions for a simulation of number of periods (`num_periods`) may take the form

```
I = 1
w = 2*pi
dt = 0.05
```

```
num_periods = 5
P = 2*pi/w    #  one period
T = P*num_periods
u, t = solver(I, w, dt, T)
visualize(u, t, I, w, dt)
```

Adjusting some of the input parameters on the command line can be handy. Here is a code segment using the `ArgumentParser` tool in the `argparse` module to define option value (`-option value`) pairs on the command line:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--I', type=float, default=1.0)
parser.add_argument('--w', type=float, default=2*pi)
parser.add_argument('--dt', type=float, default=0.05)
parser.add_argument('--num_periods', type=int, default=5)
a = parser.parse_args()
I, w, dt, num_periods = a.I, a.w, a.dt, a.num_periods
```

A typical execution goes like

```
Terminal> python vib_undamped.py --num_periods 20 --dt 0.1
```

Computing u'. In mechanical vibration applications one is often interested in computing the velocity $v(t) = u'(t)$ after $u(t)$ has been computed. This can be done by a central difference,

$$v(t_n) = u'(t_n) \approx v^n = \frac{u^{n+1} - u^{n-1}}{2\Delta t} = [D_{2t}u]^n\,. \tag{12}$$

This formula applies for all inner mesh points, $n = 1, \ldots, N_t - 1$. For $n = 0$ we have that $v(0)$ is given by the initial condition on $u'(0)$, and for $n = N_t$ we can use a one-sided, backward difference: $v^n = [D_t^- u]^n$.

Appropriate vectorized Python code becomes

```
v = np.zeros_like(u)
v[1:-1] = (u[2:] - u[:-2])/(2*dt)  # internal mesh points
v[0] = V                           # Given boundary condition u'(0)
v[-1] = (u[-1] - u[-2])/dt         # backward difference
```

.2 Verification

Manual calculation. The simplest type of verification, which is also instructive for understanding the algorithm, is to compute u^1, u^2, and u^3 with the aid of a calculator and make a function for comparing these results with those from the `solver` function. We refer to the `test_three_steps` function in the file `vib_undamped.py`[2] for details.

[2] `http://tinyurl.com/nm5587k/vib/vib_undamped.py`

Testing very simple solutions. Constructing test problems where the solution is constant or linear helps initial debugging and verification. One expects any reasonable numerical method to reproduce such solutions to machine precision. Second-order accurate methods will often also reproduce a quadratic solution. Here $[D_t D_t t^2]^n = 2$, which is the exact result. A solution $u = t^2$ fails to $u'' + \omega^2 u = 2 + (\omega t)^2 \neq 0$. We must therefore add a source in the equation, $u'' + \omega^2 u = f$ to allow a solution $u = t^2$ for $f = (\omega t)^2$. By simple inspection we can show that the mesh function $u^n = t_n^2$ is also a solution of the discrete equations. Problem 1 asks you to carry out all details with showing that linear and quadratic solutions are solutions of the discrete equations. Such results are very useful for debugging and verification.

Checking convergence rates. Empirical computation of convergence rates, as explained in Section **??**in [1], yields a good method for verification. The function below

- performs m simulations with halved time steps: $2^{-i}\Delta t$, $i = 0, \ldots,$
- computes the L^2 norm of the error, $E = \sqrt{2^{-i}\Delta t \sum_{n=0}^{N_t-1}(u^n - u_{\mathrm{e}}($ each case,
- estimates the convergence rates r_i based on two consecutive experiments $(\Delta t_{i-1}, E_{i-1})$ and $(\Delta t_i, E_i)$, assuming $E_i = C\Delta t_i^{r_i}$ and $E_{i-1} =$ From these equations it follows that $r_{i-1} = \ln(E_{i-1}/E_i)/\ln(\Delta t_{i-}$ for $i = 1, \ldots, m-1$.

All the implementational details appear below.

```
def convergence_rates(m, solver_function, num_periods=8):
    """
    Return m-1 empirical estimates of the convergence rate
    based on m simulations, where the time step is halved
    for each simulation.
    """
    w = 0.35; I = 0.3
    dt = 2*pi/w/30  # 30 time step per period 2*pi/w
    T = 2*pi/w*num_periods
    dt_values = []
    E_values = []
    for i in range(m):
        u, t = solver_function(I, w, dt, T)
        u_e = u_exact(t, I, w)
        E = sqrt(dt*sum((u_e-u)**2))
        dt_values.append(dt)
        E_values.append(E)
        dt = dt/2

    r = [log(E_values[i-1]/E_values[i])/
         log(dt_values[i-1]/dt_values[i])
         for i in range(1, m, 1)]
    return r
```

he returned `r` list has its values equal to 2.00, which is in excellent agreement ith what is expected from the second-order finite difference approximation $D_t D_t u]^n$ and other theoretical measures of the error in the numerical method. he final `r[-1]` value is a good candidate for a unit test:

```
def test_convergence_rates():
    r = convergence_rates(m=5, solver_function=solver, num_periods=8)
    # Accept rate to 1 decimal place
    nt.assert_almost_equal(r[-1], 2.0, places=1)
```

he complete code appears in the file `vib_undamped.py`.

Long time simulations

igure 1 shows a comparison of the exact and numerical solution for $\Delta t = 0.1, 0.05$ nd $w = 2\pi$. From the plot we make the following observations:

- The numerical solution seems to have correct amplitude.
- There is a phase error which is reduced by reducing the time step.
- The total phase error grows with time.

y phase error we mean that the peaks of the numerical solution have incorrect ositions compared with the peaks of the exact cosine solution. This effect an be understood as if also the numerical solution is on the form $I\cos\tilde{\omega}t$, but here $\tilde{\omega}$ is not exactly equal to ω. Later, we shall mathematically quantify this umerical frequency $\tilde{\omega}$.

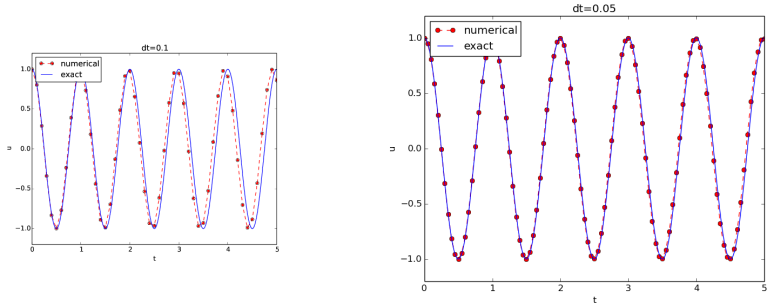


Figure 1: Effect of halving the time step.

.1 Using a moving plot window

ı vibration problems it is often of interest to investigate the system's behavior ver long time intervals. Errors in the phase may then show up as crucial. Let us

investigate long time series by introducing a moving plot window that c along with the p most recently computed periods of the solution. The S package contains a convenient tool for this: `MovingPlotWindow`. Typin `scitools.MovingPlotWindow` shows a demo and description of usag function below illustrates the usage and is invoked in the `vib_undamped.` if the number of periods in the simulation exceeds 10:

```
def visualize_front(u, t, I, w, savefig=False):
    """
    Visualize u and the exact solution vs t, using a
    moving plot window and continuous drawing of the
    curves as they evolve in time.
    Makes it easy to plot very long time series.
    """
    import scitools.std as st
    from scitools.MovingPlotWindow import MovingPlotWindow

    P = 2*pi/w  # one period
    umin = 1.2*u.min();  umax = -umin
    plot_manager = MovingPlotWindow(
        window_width=8*P,
        dt=t[1]-t[0],
        yaxis=[umin, umax],
        mode='continuous drawing')
    for n in range(1,len(u)):
        if plot_manager.plot(n):
            s = plot_manager.first_index_in_plot
            st.plot(t[s:n+1], u[s:n+1], 'r-1',
                    t[s:n+1], I*cos(w*t)[s:n+1], 'b-1',
                    title='t=%6.3f' % t[n],
                    axis=plot_manager.axis(),
                    show=not savefig) # drop window if savefig
            if savefig:
                filename = 'tmp_vib%04d.png' % n
                st.savefig(filename)
                print 'making plot file', filename, 'at t=%g' % t
        plot_manager.update(n)
```

Running

```
Terminal> python vib_undamped.py --dt 0.05 --num_periods 40
```

makes the simulation last for 40 periods of the cosine function. With the plot window we can follow the numerical and exact solution as time pr and we see from this demo that the phase error is small in the beginning, l becomes more prominent with time. Running `vib_undamped.py` with 4 clearly shows that the phase errors become significant even earlier in t series and destroys the solution.

[3] `http://code.google.com/p/scitools`

.2 Making a video

he `visualize_front` function stores all the plots in files whose names are umbered: `tmp_vib0000.png`, `tmp_vib0001.png`, `tmp_vib0002.png`, and so on. rom these files we may make a movie. The Flash format is popular,

```
erminal> avconv -r 12 -i tmp_vib%04d.png -c:v flv movie.flv
```

he `avconv` program can be replaced by the `ffmpeg` program in the above ommand if desired. The `-r` option should come first and describes the number f frames per second in the movie. The `-i` option describes the name of the plot les. Other formats can be generated by changing the video codec and equipping ie video file with the right extension:

Format	Codec and filename
Flash	`-c:v flv movie.flv`
MP4	`-c:v libx264 movie.mp4`
Webm	`-c:v libvpx movie.webm`
Ogg	`-c:v libtheora movie.ogg`

he video file can be played by some video player like `vlc`, `mplayer`, `gxine`, or `otem`, e.g.,

```
erminal> vlc movie.webm
```

web page can also be used to play the movie. Today's standard is to use the TML5 `video` tag:

```
<video autoplay loop controls
       width='640' height='365' preload='none'>
<source src='movie.webm'  type='video/webm; codecs="vp8, vorbis"'>
</video>
```

Caution: number the plot files correctly.

To ensure that the individual plot frames are shown in correct order, it is important to number the files with zero-padded numbers (0000, 0001, 0002, etc.). The printf format `%04d` specifies an integer in a field of width 4, padded with zeros from the left. A simple Unix wildcard file specification like `tmp_vib*.png` will then list the frames in the right order. If the numbers in the filenames were not zero-padded, the frame `tmp_vib11.png` would appear before `tmp_vib2.png` in the movie.

3.3 Using a line-by-line ascii plotter

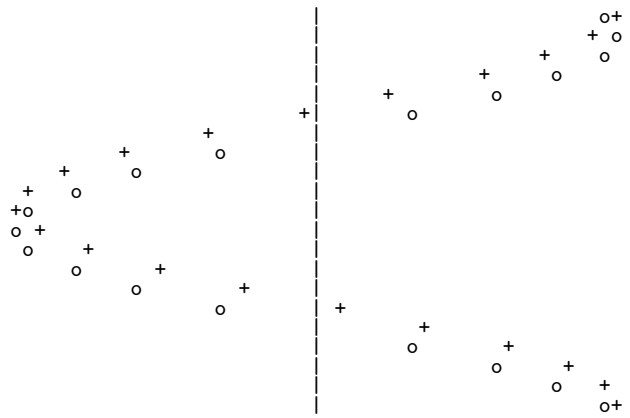
Plotting functions vertically, line by line, in the terminal window using as acters only is a simple, fast, and convenient visualization technique for lo series (the time arrow points downward). The tool `scitools.avplotter.` makes it easy to create such plots:

```
def visualize_front_ascii(u, t, I, w, fps=10):
    """
    Plot u and the exact solution vs t line by line in a
    terminal window (only using ascii characters).
    Makes it easy to plot very long time series.
    """
    from scitools.avplotter import Plotter
    import time
    P = 2*pi/w
    umin = 1.2*u.min();  umax = -umin

    p = Plotter(ymin=umin, ymax=umax, width=60, symbols='+o')
    for n in range(len(u)):
        print p.plot(t[n], u[n], I*cos(w*t[n])), \
              '%.1f' % (t[n]/P)
        time.sleep(1/float(fps))

if __name__ == '__main__':
    main()
    raw_input()
```

The call `p.plot` returns a line of text, with the t axis marked and a syml the first function (`u`) and `o` for the second function (the exact solution). append this text a time counter reflecting how many periods the curre point corresponds to. A typical output ($\omega = 2\pi$, $\Delta t = 0.05$) looks like t



3.4 Empirical analysis of the solution

For oscillating functions like those in Figure 1 we may compute the an and frequency (or period) empirically. That is, we run through the

olution points (t_n, u_n) and find all maxima and minima points. The distance etween two consecutive maxima (or minima) points can be used as estimate of ie local period, while half the difference between the u value at a maximum nd a nearby minimum gives an estimate of the local amplitude.

The local maxima are the points where

$$u^{n-1} < u^n > u^{n+1}, \quad n = 1, \ldots, N_t - 1, \tag{13}$$

nd the local minima are recognized by

$$u^{n-1} > u^n < u^{n+1}, \quad n = 1, \ldots, N_t - 1\,. \tag{14}$$

1 computer code this becomes

```
def minmax(t, u):
    minima = []; maxima = []
    for n in range(1, len(u)-1, 1):
        if u[n-1] > u[n] < u[n+1]:
            minima.append((t[n], u[n]))
        if u[n-1] < u[n] > u[n+1]:
            maxima.append((t[n], u[n]))
    return minima, maxima
```

ote that the returned objects are list of tuples.

Let (t_i, e_i), $i = 0, \ldots, M-1$, be the sequence of all the M maxima points, here t_i is the time value and e_i the corresponding u value. The local period an be defined as $p_i = t_{i+1} - t_i$. With Python syntax this reads

```
def periods(maxima):
    p = [extrema[n][0] - maxima[n-1][0]
         for n in range(1, len(maxima))]
    return np.array(p)
```

he list `p` created by a list comprehension is converted to an array since we robably want to compute with it, e.g., find the corresponding frequencies `*pi/p`.

Having the minima and the maxima, the local amplitude can be calculated s the difference between two neighboring minimum and maximum points:

```
def amplitudes(minima, maxima):
    a = [(abs(maxima[n][1] - minima[n][1]))/2.0
         for n in range(min(len(minima),len(maxima)))]
    return np.array(a)
```

he code segments are found in the file `vib_empirical_analysis.py`[4].

Visualization of the periods `p` or the amplitudes `a` it is most conveniently one with just a counter on the horizontal axis, since `a[i]` and `p[i]` correspond o the i-th amplitude estimate and the i-th period estimate, respectively. There no unique time point associated with either of these estimate since values at wo different time points were used in the computations.

In the analysis of very long time series, it is advantageous to compute and plot and `a` instead of u to get an impression of the development of the oscillations.

[4] `http://tinyurl.com/nm5587k/vib/vib_empirical_analysis.py`

4 Analysis of the numerical scheme

4.1 Deriving a solution of the numerical scheme

After having seen the phase error grow with time in the previous sect shall now quantify this error through mathematical analysis. The key the analysis will be to establish an exact solution of the discrete eq The difference equation (7) has constant coefficients and is homogeneo solution is then $u^n = CA^n$, where A is some number to be determined f differential equation and C is determined from the initial condition (Recall that n in u^n is a superscript labeling the time level, while n in A^n exponent. With oscillating functions as solutions, the algebra will be cons simplified if we seek an A on the form

$$A = e^{i\tilde{\omega}\Delta t},$$

and solve for the numerical frequency $\tilde{\omega}$ rather than A. Note that $i = \sqrt{-}$ imaginary unit. (Using a complex exponential function gives simpler arit than working with a sine or cosine function.) We have

$$A^n = e^{i\tilde{\omega}\Delta t\, n} = e^{i\tilde{\omega}t} = \cos(\tilde{\omega}t) + i\sin(\tilde{\omega}t)\,.$$

The physically relevant numerical solution can be taken as the real par complex expression.

The calculations goes as

$$\begin{aligned}
[D_tD_tu]^n &= \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} \\
&= I\frac{A^{n+1} - 2A^n + A^{n-1}}{\Delta t^2} \\
&= I\frac{\exp\left(i\tilde{\omega}(t+\Delta t)\right) - 2\exp\left(i\tilde{\omega}t\right) + \exp\left(i\tilde{\omega}(t-\Delta t)\right)}{\Delta t^2} \\
&= I\exp\left(i\tilde{\omega}t\right)\frac{1}{\Delta t^2}\left(\exp\left(i\tilde{\omega}(\Delta t)\right) + \exp\left(i\tilde{\omega}(-\Delta t)\right) - 2\right) \\
&= I\exp\left(i\tilde{\omega}t\right)\frac{2}{\Delta t^2}\left(\cosh(i\tilde{\omega}\Delta t) - 1\right) \\
&= I\exp\left(i\tilde{\omega}t\right)\frac{2}{\Delta t^2}\left(\cos(\tilde{\omega}\Delta t) - 1\right) \\
&= -I\exp\left(i\tilde{\omega}t\right)\frac{4}{\Delta t^2}\sin^2(\frac{\tilde{\omega}\Delta t}{2})
\end{aligned}$$

The last line follows from the relation $\cos x - 1 = -2\sin^2(x/2)$ (try cc in wolframalpha.com[5] to see the formula).

The scheme (7) with $u^n = Ie^{i\omega\tilde{\Delta}t\, n}$ inserted now gives

[5] `http://www.wolframalpha.com`

$$- Ie^{i\tilde{\omega}t}\frac{4}{\Delta t^2}\sin^2(\frac{\tilde{\omega}\Delta t}{2}) + \omega^2 Ie^{i\tilde{\omega}t} = 0, \tag{15}$$

hich after dividing by $Ie^{i\tilde{\omega}t}$ results in

$$\frac{4}{\Delta t^2}\sin^2(\frac{\tilde{\omega}\Delta t}{2}) = \omega^2 . \tag{16}$$

he first step in solving for the unknown $\tilde{\omega}$ is

$$\sin^2(\frac{\tilde{\omega}\Delta t}{2}) = \left(\frac{\omega\Delta t}{2}\right)^2 .$$

hen, taking the square root, applying the inverse sine function, and multiplying y $2/\Delta t$, results in

$$\tilde{\omega} = \pm\frac{2}{\Delta t}\sin^{-1}\left(\frac{\omega\Delta t}{2}\right) . \tag{17}$$

The first observation of (17) tells that there is a phase error since the umerical frequency $\tilde{\omega}$ never equals the exact frequency ω. But how good is ıe approximation (17)? That is, what is the error $\omega - \tilde{\omega}$ or $\tilde{\omega}/\omega$? Taylor series xpansion for small Δt may give an expression that is easier to understand than ıe complicated function in (17):

```
>>> from sympy import *
>>> dt, w = symbols('dt w')
>>> w_tilde_e = 2/dt*asin(w*dt/2)
>>> w_tilde_series = w_tilde_e.series(dt, 0, 4)
>>> print w_tilde_series
w + dt**2*w**3/24 + O(dt**4)
```

his means that

$$\tilde{\omega} = \omega\left(1 + \frac{1}{24}\omega^2\Delta t^2\right) + \mathcal{O}(\Delta t^4) . \tag{18}$$

he error in the numerical frequency is of second-order in Δt, and the error anishes as $\Delta t \to 0$. We see that $\tilde{\omega} > \omega$ since the term $\omega^3\Delta t^2/24 > 0$ and this by far the biggest term in the series expansion for small $\omega\Delta t$. A numerical equency that is too large gives an oscillating curve that oscillates too fast and ıerefore "lags behind" the exact oscillations, a feature that can be seen in the lots.

Figure 2 plots the discrete frequency (17) and its approximation (18) for = 1 (based on the program `vib_plot_freq.py`[6]). Although $\tilde{\omega}$ is a function f Δt in (18), it is misleading to think of Δt as the important discretization arameter. It is the product $\omega\Delta t$ that is the key discretization parameter. This uantity reflects the *number of time steps per period* of the oscillations. To see ıis, we set $P = N_P\Delta t$, where P is the length of a period, and N_P is the number

[6] `http://tinyurl.com/nm5587k/vib/vib_plot_freq.py`

of time steps during a period. Since P and ω are related by $P = 2\pi/\omega$ that $\omega\Delta t = 2\pi/N_P$, which shows that $\omega\Delta t$ is directly related to N_P.

The plot shows that at least $N_P \sim 25 - 30$ points per period are n for reasonable accuracy, but this depends on the length of the simulatio the total phase error due to the frequency error grows linearly with ti Exercise 2).

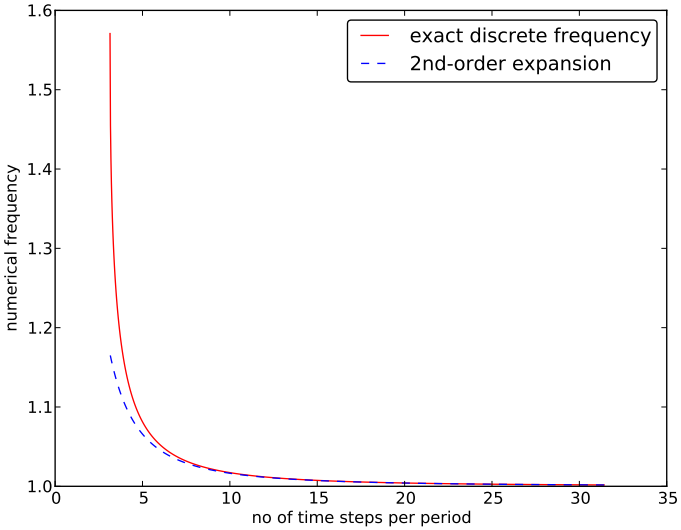


Figure 2: Exact discrete frequency and its second-order series expan

4.2 Exact discrete solution

Perhaps more important than the $\tilde{\omega} = \omega + \mathcal{O}(\Delta t^2)$ result found above is that we have an exact discrete solution of the problem:

$$u^n = I\cos(\tilde{\omega}n\Delta t) , \quad \tilde{\omega} = \frac{2}{\Delta t}\sin^{-1}\left(\frac{\omega\Delta t}{2}\right) .$$

We can then compute the error mesh function

$$e^n = u_e(t_n) - u^n = I\cos(\omega n\Delta t) - I\cos(\tilde{\omega}n\Delta t) .$$

From the formula $\cos 2x - \cos 2y = -2\sin(x-y)\sin(x+y)$ we can rewri the expression is easier to interpret:

$$e^n = -2I\sin\left(t\frac{1}{2}(\omega - \tilde{\omega})\right)\sin\left(t\frac{1}{2}(\omega + \tilde{\omega})\right) .$$

In particular, we can use (21) to show *convergence* of the numerical scheme, e., $e^n \to 0$ as $\Delta t \to 0$. We have that

$$\lim_{\Delta t\to 0} \tilde{\omega} = \lim_{\Delta t\to 0} \frac{2}{\Delta t}\sin^{-1}\left(\frac{\omega\Delta t}{2}\right) = \omega,$$

y L'Hopital's rule or simply asking `sympy` or WolframAlpha[7] about the limit:

```
>>> import sympy as sp
>>> dt, w = sp.symbols('x w')
>>> sp.limit((2/dt)*sp.asin(w*dt/2), dt, 0, dir='+')
w
```

herefore, $\tilde{\omega} \to \omega$ and obviously $e^n \to 0$.

The error mesh function is ideal for verification purposes and you are strongly ncouraged to make a test based on (19) by doing Exercise 10.

.3 The global error

o achieve more analytical insight into the nature of the global error, we can aylor expand the error mesh function. Since $\tilde{\omega}$ contains Δt in the denominator e use the series expansion for $\tilde{\omega}$ inside the cosine function:

```
>>> dt, w, t = symbols('dt w t')
>>> w_tilde_e = 2/dt*asin(w*dt/2)
>>> w_tilde_series = w_tilde_e.series(dt, 0, 4)
>>> # Get rid of O() term
>>> w_tilde_series = sum(w_tilde_series.as_ordered_terms()[:-1])
>>> w_tilde_series
dt**2*w**3/24 + w
>>> error = cos(w*t) - cos(w_tilde_series*t)
>>> error.series(dt, 0, 6)
dt**2*t*w**3*sin(t*w)/24 + dt**4*t**2*w**6*cos(t*w)/1152 + O(dt**6)
>>> error.series(dt, 0, 6).as_leading_term(dt)
dt**2*t*w**3*sin(t*w)/24
```

his means that the leading order global (true) error at a point t is proportional o $\omega^3 t\Delta t^2$. Setting $t = n\Delta t$ and replacing $\sin(\omega t)$ by its maximum value 1, we ave the analytical leading-order expression

$$e^n = \frac{1}{24}n\omega^3\Delta t^3,$$

nd the ℓ^2 norm of this error can be computed as

$$||e^n||^2_{\ell^2} = \Delta t\sum_{n=0}^{N_t}\frac{1}{24^2}n^2\omega^6\Delta t^6 = \frac{1}{24^2}\omega^6\Delta t^7\sum_{n=0}^{N_t} n^2\,.$$

he sum $\sum_{n=0}^{N_t} n^2$ is approximately equal to $\frac{1}{3}N_t^3$. Replacing N_t by $T/\Delta t$ and aking the square root gives the expression

[7] `http://www.wolframalpha.com/input/?i=%282%2Fx%29*asin%28w*x%2F2%29+as+x-%3E0`

$$||e^n||_{\ell^2} = \frac{1}{24}\sqrt{\frac{T^3}{3}}\omega^3\Delta t^2,$$

which shows that also the integrated error is proportional to Δt^2.

4.4 Stability

Looking at (19), it appears that the numerical solution has constant and amplitude, but an error in the frequency (phase error). However, there is error that is more serious, namely an unstable growing amplitude that ca of Δt is too large.

We realize that a constant amplitude demands $\tilde{\omega}$ to be a real nun complex $\tilde{\omega}$ is indeed possible if the argument x of $\sin^{-1}(x)$ has magnitud than unity: $|x| > 1$ (type `asin(x)` in wolframalpha.com[8] to see basic prop $\sin^{-1}(x)$). A complex $\tilde{\omega}$ can be written $\tilde{\omega} = \tilde{\omega}_r + i\tilde{\omega}_i$. Since $\sin^{-1}(x)$ has a imaginary part for $x > 1$, $\tilde{\omega}_i < 0$, it means that $\exp(i\omega\tilde{t}) = \exp(-\tilde{\omega}_i t)$ ex will lead to exponential growth in time because $\exp(-\tilde{\omega}_i t)$ with $\tilde{\omega}_i <$ positive exponent.

We do not tolerate growth in the amplitude and we therefore have a *criterion* arising from requiring the argument $\omega\Delta t/2$ in the inverse sine f to be less than one:

$$\frac{\omega\Delta t}{2} \leq 1 \quad \Rightarrow \quad \Delta t \leq \frac{2}{\omega}\,.$$

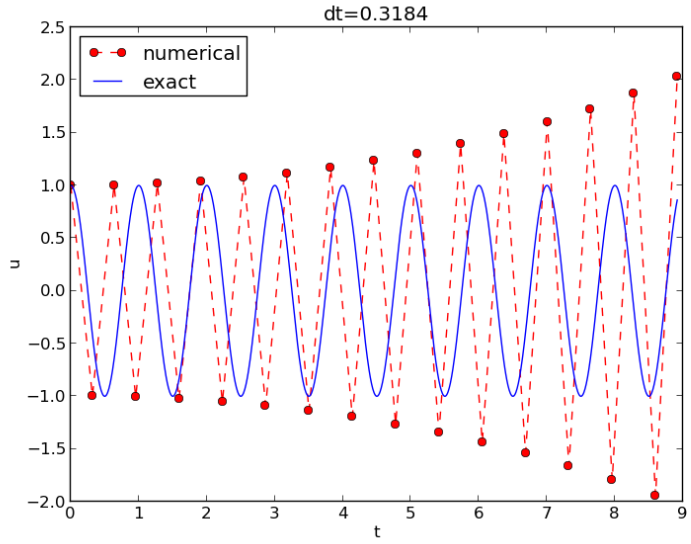
With $\omega = 2\pi$, $\Delta t > \pi^{-1} = 0.3183098861837907$ will give growing so Figure 3 displays what happens when $\Delta t = 0.3184$, which is slightly ab critical value: $\Delta t = \pi^{-1} + 9.01\cdot 10^{-5}$.

4.5 About the accuracy at the stability limit

An interesting question is whether the stability condition $\Delta t < 2/\omega$ is unfo or more precisely: would it be meaningful to take larger time steps t up computations? The answer is a clear no. At the stability limit, v that $\sin^{-1}\omega\Delta t/2 = \sin^{-1} 1 = \pi/2$, and therefore $\tilde{\omega} = \pi/\Delta t$. (Note t approximate formula (18) is very inaccurate for this value of Δt as it $\tilde{\omega} = 2.34/pi$, which is a 25 percent reduction.) The corresponding perio numerical solution is $\tilde{P} = 2\pi/\tilde{\omega} = 2\Delta t$, which means that there is just c step Δt between a peak and a through in the numerical solution. Th shortest possible wave that can be represented in the mesh. In other wo not meaningful to use a larger time step than the stability limit.

Also, the phase error when $\Delta t = 2/\omega$ is severe: Figure 4 shows a com of the numerical and analytical solution with $\omega = 2\pi$ and $\Delta t = 2/\omega$ Already after one period, the numerical solution has a through while th solution has a peak (!). The error in frequency when Δt is at the stabil becomes $\omega - \tilde{\omega} = \omega(1 - \pi/2) \approx -0.57\omega$. The corresponding error in the

[8] `http://www.wolframalpha.com`



igure 3: Growing, unstable solution because of a time step slightly beyond ıe stability limit.

$P - \tilde{P} \approx 0.36P$. The error after m periods is then $0.36mP$. This error has ɔach half a period when $m = 1/(2 \cdot 0.36) \approx 1.38$, which theoretically confirms ıe observations in Figure 4 that the numerical solution is a through ahead of a eak already after one and a half period.

Summary.

From the accuracy and stability analysis we can draw three important conclusions:

1. The key parameter in the formulas is $p = \omega\Delta t$. The period of oscillations is $P = 2\pi/\omega$, and the number of time steps per period is $N_P = P/\Delta t$. Therefore, $p = \omega\Delta t = 2\pi N_P$, showing that the critical parameter is the number of time steps per period. The smallest possible N_P is 2, showing that $p \in (0, \pi]$.
2. Provided $p \leq 2$, the amplitude of the numerical solution is constant.
3. The numerical solution exhibits a relative phase error $\tilde{\omega}/\omega \approx 1 + \frac{1}{24}p^2$. This error leads to wrongly displaced peaks of the numerical

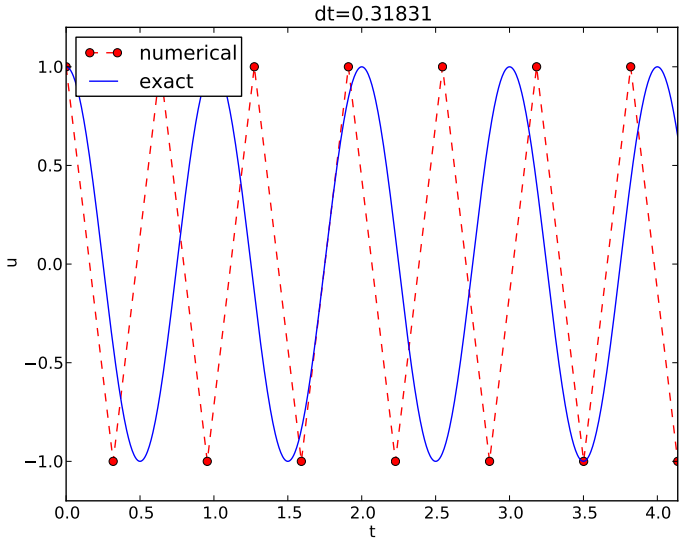


Figure 4: Numerical solution with Δt exactly at the stability limi

solution, and the error in peak location grows linearly with time Exercise 2).

5 Alternative schemes based on 1st-order tions

A standard technique for solving second-order ODEs is to rewrite the system of first-order ODEs and then apply the vast collection of meth first-order ODE systems. Given the second-order ODE problem

$$u'' + \omega^2 u = 0, \quad u(0) = I, \ u'(0) = 0,$$

we introduce the auxiliary variable $v = u'$ and express the ODE pro terms of first-order derivatives of u and v:

$$u' = v,$$
$$v' = -\omega^2 u\,.$$

The initial conditions become $u(0) = I$ and $v(0) = 0$.

Standard methods for 1st-order ODE systems

.1 The Forward Euler scheme

Forward Euler approximation to our 2×2 system of ODEs (23)-(24) becomes

$$[D_t^+ u = v]^n, [D_t^+ v = -\omega^2 u]^n, \tag{25}$$

r written out,

$$u^{n+1} = u^n + \Delta t v^n, \tag{26}$$
$$v^{n+1} = v^n - \Delta t \omega^2 u^n \,. \tag{27}$$

Let us briefly compare this Forward Euler method with the centered difference cheme for the second-order differential equation. We have from (26) and (27) pplied at levels n and $n-1$ that

$$u^{n+1} = u^n + \Delta t v^n = u^n + \Delta t (v^{n-1} - \Delta t \omega^2 u^{n-1} \,.$$

ince from (26)

$$v^{n-1} = \frac{1}{\Delta t}(u^n - u^{n-1}),$$

follows that

$$u^{n+1} = 2u^n - u^{n-1} - \Delta t^2 \omega^2 u^{n-1},$$

hich is very close to the centered difference scheme, but the last term is evaluated t t_{n-1} instead of t_n. Dividing by Δt^2, the left-hand side is an approximation u'' at t_n, while the right-hand side is sampled at t_{n-1}. This inconsistency in ıe scheme turns out to be rather crucial for the accuracy of the Forward Euler ıethod applied to vibration problems.

.2 The Backward Euler scheme

Backward Euler approximation the ODE system is equally easy to write up in ıe operator notation:

$$[D_t^- u = v]^{n+1}, \tag{28}$$
$$[D_t^- v = -\omega u]^{n+1} \,. \tag{29}$$

his becomes a coupled system for u^{n+1} and v^{n+1}:

$$u^{n+1} - \Delta t v^{n+1} = u^n, \tag{30}$$
$$v^{n+1} + \Delta t \omega^2 u^{n+1} = v^n \,. \tag{31}$$

We can compare (30)-(31) with central the scheme for the secon differential equation. To this end, we eliminate v^{n+1} in (30) using (31 with respect to v^{n+1}. Thereafter, we eliminate v^n using (30) solved with to v^{n+1} and replacing $n+1$ by n. The resulting equation involving on u^n, and u^{n-1} can be ordered as

$$\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} = -\omega^2 u^{n+1},$$

which has almost the same form as the centered scheme for the secon differential equation, but the right-hand side is evaluated at u^{n+1} and This obvious inconsistency has a dramatic effect on the numerical solut

6.3 The Crank-Nicolson scheme

The Crank-Nicolson scheme takes this form in the operator notation:

$$[D_t u = \overline{v}^t]^{n+\frac{1}{2}},$$
$$[D_t v = -\omega \overline{u}^t]^{n+\frac{1}{2}} \,.$$

Writing the equations out shows that this is also a coupled system:

$$u^{n+1} - \frac{1}{2}\Delta t v^{n+1} = u^n + \frac{1}{2}\Delta t v^n,$$
$$v^{n+1} + \frac{1}{2}\Delta t \omega^2 u^{n+1} = v^n - \frac{1}{2}\Delta t \omega^2 u^n \,.$$

6.4 Comparison of schemes

We can easily compare methods like the ones above (and many more!) v aid of the Odespy[9] package. Below is a sketch of the code.

```
import odespy
import numpy as np

def f(u, t, w=1):
    u, v = u  # u is array of length 2 holding our [u, v]
    return [v, -w**2*u]

def run_solvers_and_plot(solvers, timesteps_per_period=20,
                         num_periods=1, I=1, w=2*np.pi):
    P = 2*np.pi/w  # duration of one period
    dt = P/timesteps_per_period
    Nt = num_periods*timesteps_per_period
    T = Nt*dt
    t_mesh = np.linspace(0, T, Nt+1)

    legends = []
```

[9] https://github.com/hplgit/odespy

```
    for solver in solvers:
        solver.set(f_kwargs={'w': w})
        solver.set_initial_condition([I, 0])
        u, t = solver.solve(t_mesh)
```

here is quite some more code dealing with plots also, and we refer to the source le `vib_undamped_odespy.py`[10] for details. Observe that keyword arguments in `(u,t,w=1)` can be supplied through a solver parameter `f_kwargs` (dictionary f additional keyword arguments to `f`).

Specification of the Forward Euler, Backward Euler, and Crank-Nicolson chemes is done like this:

```
solvers = [
    odespy.ForwardEuler(f),
    # Implicit methods must use Newton solver to converge
    odespy.BackwardEuler(f, nonlinear_solver='Newton'),
    odespy.CrankNicolson(f, nonlinear_solver='Newton'),
    ]
```

The `vib_undamped_odespy.py` program makes two plots of the computed olutions with the various methods in the `solvers` list: one plot with $u(t)$ versus and one *phase plane plot* where v is plotted against u. That is, the phase lane plot is the curve $(u(t), v(t))$ parameterized by t. Analytically, $u = I\cos(\omega t)$ nd $v = u' = -\omega I\sin(\omega t)$. The exact curve $(u(t), v(t))$ is therefore an ellipse, hich often looks like a circle in a plot if the axes are automatically scaled. he important feature, however, is that exact curve $(u(t), v(t))$ is closed and epeats itself for every period. Not all numerical schemes are capable to do that, eaning that the amplitude instead shrinks or grows with time.

The Forward Euler scheme in Figure 5 has a pronounced spiral curve, pointing o the fact that the amplitude steadily grows, which is also evident in Figure 6. he Backward Euler scheme has a similar feature, except that the spriral goes ward and the amplitude is significantly damped. The changing amplitude nd the sprial form decreases with decreasing time step. The Crank-Nicolson cheme looks much more accurate. In fact, these plots tell that the Forward and ackward Euler schemes are not suitable for solving our ODEs with oscillating olutions.

.5 Runge-Kutta methods

/e may run two popular standard methods for first-order ODEs, the 2nd- and ch-order Runge-Kutta methods, to see how they perform. Figures 7 and 8 show ne solutions with larger Δt values than what was used in the previous two plots.

The visual impression is that the 4th-order Runge-Kutta method is very ccurate, under all circumstances in these tests, and the 2nd-order scheme suffer om amplitude errors unless the time step is very small.

The corresponding results for the Crank-Nicolson scheme are shown in igures 9 and 10. It is clear that the Crank-Nicolson scheme outperforms

[10] `http://tinyurl.com/nm5587k/vib/vib_undamped_odespy.py`

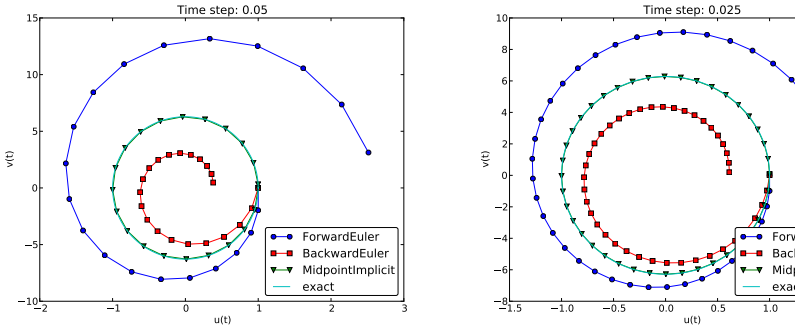


Figure 5: Comparison of classical schemes in the phase plane.

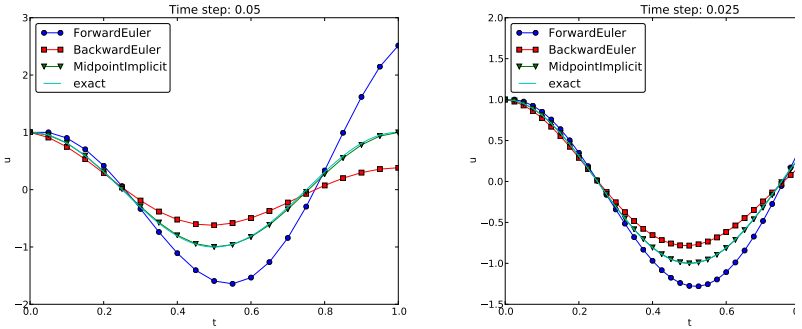


Figure 6: Comparison of classical schemes.

the 2nd-order Runge-Kutta method. Both schemes have the same c accuracy $\mathcal{O}(\Delta t^2)$, but their differences in the accuracy that matters i physical application is very clearly pronounced in this example. Exe invites you to investigate how

6.6 Analysis of the Forward Euler scheme

We may try to find exact solutions of the discrete equations in the Forwa method. An "ansatz" is

$$u^n = IA^n,$$
$$v^n = qIA^n,$$

where q and A are unknown numbers. We could have used a complex exp form $\exp(i\tilde{\omega}n\Delta t)$ since we get oscillatory form, but the oscillations gro Forward Euler method, so the numerical frequency $\tilde{\omega}$ will be complex (to produce an exponentially growing amplitude), so it is easier to ju with potentially complex A and q as introduced above.

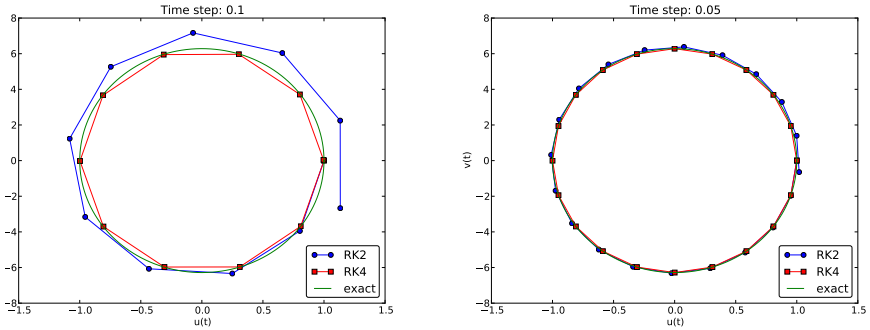


Figure 7: Comparison of Runge-Kutta schemes in the phase plane.

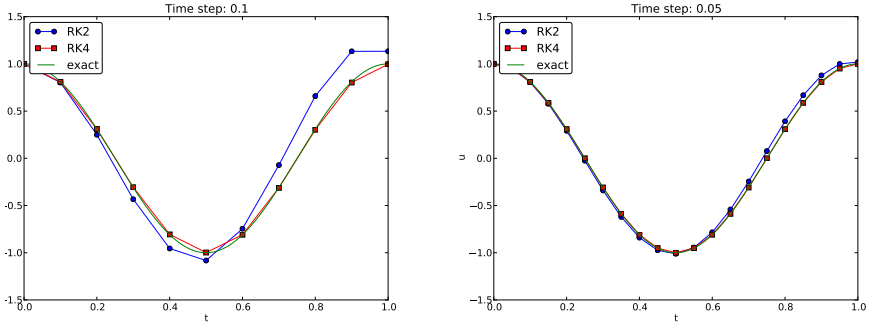


Figure 8: Comparison of Runge-Kutta schemes.

The Forward Euler scheme leads to

$$A = 1 + \Delta t q,$$
$$A = 1 - \Delta t \omega^2 q^{-1}.$$

We can easily eliminate A, get $q^2 + \omega^2 = 0$, and solve for

$$q = \pm i\omega,$$

which gives

$$A = 1 \pm \Delta t i\omega\,.$$

We shall take the real part of A^n as the solution. The two values of A are complex conjugates, and the real part of A^n will be the same for the two roots. This is easy to realize if we rewrite the complex numbers in polar form ($re^{i\theta}$), which is also convenient for further analysis and understanding. The polar form of the two values for A become

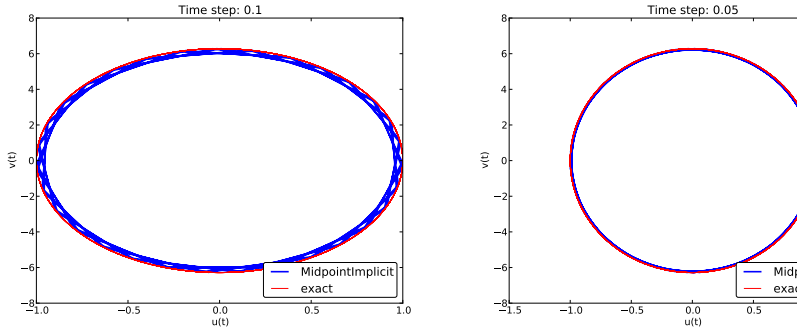


Figure 9: Long-time behavior of the Crank-Nicolson scheme in the pha

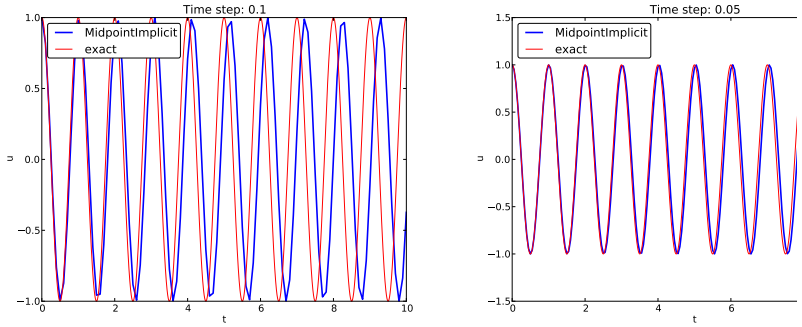


Figure 10: Long-time behavior of the Crank-Nicolson scheme.

$$1 \pm \Delta t i\omega = \sqrt{1 + \omega^2 \Delta t^2} e^{\pm i \tan^{-1}(\omega \Delta t)}\,.$$

Now,

$$(1 \pm \Delta t i\omega)^n = (1 + \omega^2 \Delta t^2) n/2 e^{\pm n i \tan^{-1}(\omega \Delta t)}\,.$$

Since $\cos(\theta n) = \cos(-\theta n)$, the real part of the two numbers become th We therefore continue with the solution that has the plus sign.

The general solution is $u^n = CA^n$, where C is a constant. This is det from the initial condition: $u^0 = C = I$. Then also $v^n = qIA^n$. T solutions consist of the real part of the expressions in polar form:

$$u^n = I(1+\omega^2\Delta t^2)n/2 \cos(n \tan^{-1}(\omega \Delta t)), \quad v^n = -\omega I(1+\omega^2\Delta t^2)n/2 \sin$$

The expression $(1 + \omega^2 \Delta t^2)n/2$ causes growth of the amplitude, since a greater than one is raised to an integer power. By expanding first the root, $\sqrt{1 + x^2} \approx 1 + \frac{1}{2}x^2$, we realize that raising the approximation to any

ower, will give rise to a polynomial with leading terms $1 + x^2$, or with $x = \omega\Delta$ s in our case, the amplitude in the Forward Euler scheme grows as $1 + \omega^2\Delta t^2$.

Energy considerations

he observations of various methods in the previous section can be better nterpreted if we compute an quantity reflecting the total *energy of the system.* turns out that this quantity,

$$E(t) = \frac{1}{2}(u')^2 + \frac{1}{2}\omega^2 u^2,$$

constant for all t. Checking that $E(t)$ really remains constant brings evidence hat the numerical computations are sound. Such energy measures, when they xist, are much used to check numerical simulations.

.1 Derivation of the energy expression

/e starting multiplying

$$u'' + \omega^2 u = 0,$$

y u' and integrating from 0 to T:

$$\int_0^T u''u'dt + \int_0^T \omega^2 uu'dt = 0\,.$$

bserving that

$$u''u' = \frac{d}{dt}\frac{1}{2}(u')^2, \quad uu' = \frac{d}{dt}\frac{1}{2}u^2,$$

e get

$$\int_0^T (\frac{d}{dt}\frac{1}{2}(u')^2 + \frac{d}{dt}\frac{1}{2}\omega^2 u^2)dt = E(T) - E(0) = 0,$$

here we have introduced the energy measure $E(t)$

$$E(t) = \frac{1}{2}(u')^2 + \frac{1}{2}\omega^2 u^2\,. \tag{36}$$

he important result from this derivation is that the total energy is constant:

$$E(t) = E(0)\,.$$

Warning.

The quantity $E(t)$ derived above is physically not the energy of a vibrating mechanical system, but the energy per unit mass. To see this, we start

with Newton's second law $F = ma$ (F is the sum of forces, m is the of the system, and a is the acceleration). The displacement u is relat a through $a = u''$. With a spring force as the only force we have $F =$ where k is a spring constant measuring the stiffness of the spring. New second law then implies the differential equation

$$-ku = mu'' \quad \Rightarrow mu'' + ku = 0\,.$$

This equation of motion can be turned into an energy balance equatic finding the work done by each term during a time interval $[0, T]$. To end, we multiply the equation by $du = u'dt$ and integrate:

$$\int_0^T muu'dt + \int_0^T kuu'dt = 0\,.$$

The result is

$$E(t) = E_k(t) + E_p(t) = 0,$$

where

$$E_k(t) = \frac{1}{2}mv^2, \quad v = u',$$

is the *kinetic energy* of the system,

$$E_p(t) = \frac{1}{2}ku^2$$

is the *potential energy*, and the sum $E(t)$ is the total energy. The deriv demonstrates the famous energy principle that any change in the ki energy is due to a change in potential energy and vice versa.

The equation $mu'' + ku = 0$ can be divided by m and writte $u'' + \omega^2 u = 0$ for $\omega = \sqrt{k/m}$. The energy expression $E(t) = \frac{1}{2}(u')^2 + \frac{1}{2}$ derived earlier is then simply the true physical total energy $\frac{1}{2}m(u')^2 + \frac{1}{2}$ divided by m, i.e., total energy per unit mass.

Energy of the exact solution. Analytically, we have $u(t) = I$ c $u(0) = I$ and $u'(0) = 0$, so we can easily check that the evolution of the $E(t)$ is constant:

$$E(t) = \frac{1}{2}I^2(-\omega\sin\omega t)^2 + \frac{1}{2}\omega^2 I^2\cos^2\omega t = \frac{1}{2}\omega^2(\sin^2\omega t + \cos^2\omega t) =$$

7.2 An error measure based on total energy

The error in total energy, as a mesh function, can be computed by

$$e_E^n = \frac{1}{2}\left(\frac{u^{n+1}-u^{n-1}}{2\Delta t}\right)^2 + \frac{1}{2}\omega^2(u^n)^2 - E(0), \quad n = 1,\ldots,N_t-1, \qquad (39)$$

here

$$E(0) = \frac{1}{2}V^2 + \frac{1}{2}\omega^2 I^2,$$

$u(0) = I$ and $u'(0) = V$. A useful norm can be the maximum absolute value f e_E^n:

$$||e_E^n||_{\ell^\infty} = \max_{1\le n<N_t} |e_E^n|\,.$$

he corresponding Python implementation takes the form

```
# import numpy as np and compute u, t
dt = t[1]-t[0]
E = 0.5*((u[2:] - u[:-2])/(2*dt))**2 + 0.5*w**2*u[1:-1]**2
E0 = 0.5*V**2 + 0.5**w**2*I**2
e_E = E - E0
e_E_norm = np.abs(e_E).max()
```

The convergence rates of the quantity `e_E_norm` can be used for verification. he value of `e_E_norm` is also useful for comparing schemes through their ability o preserve energy. Below is a table demonstrating the error in total energy or various schemes. We clearly see that the Crank-Nicolson and 4th-order unge-Kutta schemes are superior to the 2nd-order Runge-Kutta method and ven more superior to the Forward and Backward Euler schemes.

Method	T	Δt	$\max\|e_E^n\|$
Forward Euler	1	0.05	$1.113\cdot 10^{2}$
Forward Euler	1	0.025	$3.312\cdot 10^{1}$
Backward Euler	1	0.05	$1.683\cdot 10^{1}$
Backward Euler	1	0.025	$1.231\cdot 10^{1}$
Runge-Kutta 2nd-order	1	0.1	8.401
Runge-Kutta 2nd-order	1	0.05	$9.637\cdot 10^{-1}$
Crank-Nicolson	1	0.05	$9.389\cdot 10^{-1}$
Crank-Nicolson	1	0.025	$2.411\cdot 10^{-1}$
Runge-Kutta 4th-order	1	0.1	2.387
Runge-Kutta 4th-order	1	0.05	$6.476\cdot 10^{-1}$
Crank-Nicolson	10	0.1	3.389
Crank-Nicolson	10	0.05	$9.389\cdot 10^{-1}$
Runge-Kutta 4th-order	10	0.1	3.686
Runge-Kutta 4th-order	10	0.05	$6.928\cdot 10^{-1}$

8 The Euler-Cromer method

While the 4th-order Runge-Kutta method and the a centered Crank-N scheme work well for the first-order formulation of the vibration model, bo inferior to the straightforward centered difference scheme for the secon equation $u'' + \omega^2 u = 0$. However, there is a similarly successful scheme a for the first-order system $u' = v$, $v' = -\omega^2 u$, to be presented next.

8.1 Forward-backward discretization

The idea is to apply a Forward Euler discretization to the first equatio Backward Euler discretization to the second. In operator notation this i as

$$[D_t^+ u = v]^n,$$
$$[D_t^- v = -\omega u]^{n+1}\,.$$

We can write out the formulas and collect the unknowns on the left-ha

$$u^{n+1} = u^n + \Delta t v^n,$$
$$v^{n+1} = v^n - \Delta t\omega^2 u^{n+1}\,.$$

We realize that u^{n+1} can be computed from (42) and then v^{n+1} from (4 the recently computed value u^{n+1} on the right-hand side.

In physics, it is more common to update the v equation first, with a difference, and thereafter the u equation, with a backward difference that the most recently computed v value:

$$v^{n+1} = v^n - \Delta t\omega^2 u^n,$$
$$u^{n+1} = u^n + \Delta t v^{n+1}\,.$$

The advantage of this sequence of the first-order ODEs becomes evide we turn to more complicated models. A typical vibration ODE can in be written as

$$\ddot{u} + g(u, u', t) = 0,$$

which results in the system

$$v' = -g(u, v, t),$$
$$u' = v,$$

and the scheme

$$v^{n+1} = v^n - \Delta t\, g(u^n, v^n, t),$$
$$u^{n+1} = u^n + \Delta t\, v^{n+1}\,.$$

/e realize that the first update works well with any g since old values u^n and ^{n} are used. Switching the equations would demand $u^n{+}1$ and v^{n+1} values in g.

The scheme (45)-(44) goes under several names: Forward-backward scheme, emi-implicit Euler method[11], symplectic Euler, semi-explicit Euler, Newton-törmer-Verlet, and Euler-Cromer. We shall stick to the latter name. Since both me discretizations are based on first-order difference approximation, one may ink that the scheme is only of first-order, but this is not true: the use of a orward and then a backward difference make errors cancel so that the overall ror in the scheme is $\mathcal{O}(\Delta t^2)$. This is explaned below.

.2 Equivalence with the scheme for the second-order ODE

/e may eliminate the v^n variable from (42)-(43) or (45)-(44). The v^{n+1} term in 4) can be eliminated from (45):

$$u^{n+1} = u^n + \Delta t(v^n - \omega^2\Delta t^2 u^n)\,. \tag{46}$$

he v^n quantity can be expressed by u^n and u^{n-1} using (45):

$$v^n = \frac{u^n - u^{n-1}}{\Delta t},$$

nd when this is inserted in (46) we get

$$u^{n+1} = 2u^n - u^{n-1} - \Delta t^2\omega^2 u^n, \tag{47}$$

hich is nothing but the centered scheme (7)! Therefore, the previous analysis f (7) also applies to the Euler-Cromer method. In particular, the amplitude constant, given that the stability criterion is fulfilled, but there is always a hase error (18). Exercise 20 gives guidance on how to derive the exact discrete olution of the two equations in the Euler-Cromer method.

The initial condition $u' = 0$ means $u' = v = 0$. From (45) we get $v^1 = -\omega^2 u^0$ nd $u^1 = u^0 - \omega^2\Delta t^2 u^0$, which is not exactly the same u^1 value as obtained by centered approximation of $v'(0) = 0$ and combined with the discretization (7) f the second-order ODE: a factor $\frac{1}{2}$ is missing in the second term. In fact, if e approximate $u'(0) = 0$ by a backward difference, $(u^0 - u^{-1})/\Delta t = 0$, we get $^{-1} = u^0$, and when combined with (7), it results in $u^1 = u^0 - \omega^2\Delta t^2 u^0$. That , the Euler-Cromer method based on (45)-(44) corresponds to using only a rst-order approximation to the initial condition in the method from Section 1.2.

Correspondingly, using the formulation (42)-(43) with $v^n = 0$ leads to $u^1 = u^0$, hich can be interpreted as using a forward difference approximation for the itial condition $u'(0) = 0$. Both Euler-Cromer formulations lead to slightly ifferent values for u^1 compared to the method in Section 1.2. The error is $\omega^2\Delta t^2 u^0$ and of the same order as the overall scheme.

[11] http://en.wikipedia.org/wiki/Semi-implicit_Euler_method

8.3 Implementation

The function below, found in `vib_EulerCromer.py`[12] implements th Cromer scheme (45)-(44):

```
from numpy import zeros, linspace

def solver(I, w, dt, T):
    """
    Solve v' = - w**2*u, u'=v for t in (0,T], u(0)=I and v(0)=0,
    by an Euler-Cromer method.
    """
    dt = float(dt)
    Nt = int(round(T/dt))
    u = zeros(Nt+1)
    v = zeros(Nt+1)
    t = linspace(0, Nt*dt, Nt+1)

    v[0] = 0
    u[0] = I
    for n in range(0, Nt):
        v[n+1] = v[n] - dt*w**2*u[n]
        u[n+1] = u[n] + dt*v[n+1]
    return u, v, t
```

Since the Euler-Cromer scheme is equivalent to the finite difference for the second-order ODE $u'' + \omega^2 u = 0$ (see Section 8.2), the performanc above `solver` function is the same as for the `solver` function in Section only difference is the formula for the first time step, as discussed abov deviation in the Euler-Cromer scheme means that the discrete solution Section 4.2 is not a solution of the Euler-Cromer scheme. To verify th mentation of the Euler-Cromer method we can adjust `v[1]` so that the cc generated values can be compared formula from in Section 4.2. This adjus done in an alternative solver function, `solver_ic_fix` in `vib_EulerCrc` and combined with a nose test in the function `test_solver` that check ity of computed values with the exact discrete solution to machine p Another function, `demo`, visualizes the difference between Euler-Cromer and the scheme for the second-oder ODE, arising from the mismatch in time level.

8.4 The velocity Verlet algorithm

Another very popular algorithm for vibration problems $u'' + \omega^2 u = 0$ derived as follows. First, we step u forward from t_n to t_{n+1} using a thr Taylor series,

$$u(t_{n+1}) = u(t_n) + u'(t_n)\Delta t + \frac{1}{2}u''(t_n)\Delta t^2\,.$$

Using $u' = v$ and $u'' = -\omega^2 u$, we get the updating formula

[12] http://tinyurl.com/nm5587k/vib/vib_EulerCromer.py

$$u^{n+1} = u^n + v^n\Delta t - \frac{1}{2}\Delta^2\omega^2 u^n\,.$$

econd, the first-order equation for v,

$$v' = -\omega^2 u,$$

discretized by a centered difference in a Crank-Nicolson fashion at $t_{n+\frac{1}{2}}$:

$$\frac{v^{n+1}-v^n}{\Delta t} = -\omega^2\frac{1}{2}(u^n + u^{n+1}),$$

r in operator form that explicitly demonstrates the thinking:

$$[D_t u = -\omega^2\bar{u}^t]^{n+\frac{1}{2}}\,.$$

o summarize, we have the scheme

$$u^{n+1} = u^n + v^n\Delta t - \frac{1}{2}\Delta^2\omega^2 u^n \tag{48}$$

$$v^{n+1} = v^n - \frac{1}{2}\Delta t\omega^2(u^n + u^{n+1}), \tag{49}$$

nown as the *velocity Verlet* algorithm. Observe that this scheme is explicit nce u^{n+1} in (49) is already computed from (48).

The algorithm can be straightforwardly implemented as shown below.

```
rom vib_undamped import (
    zeros, linspace,
    convergence_rates,
    main)

lef solver(I, w, dt, T, return_v=False):
    """
    Solve u'=v, v'=-w**2*u for t in (0,T], u(0)=I and v(0)=0,
    by the velocity Verlet method with time step dt.
    """
    dt = float(dt)
    Nt = int(round(T/dt))
    u = zeros(Nt+1)
    v = zeros(Nt+1)
    t = linspace(0, Nt*dt, Nt+1)

    u[0] = I
    v[0] = 0
    for n in range(Nt):
        u[n+1] = u[n] + v[n]*dt - 0.5*dt**2*w**2*u[n]
        v[n+1] = v[n] - 0.5*dt*w**2*(u[n] + u[n+1])
    if return_v:
        return u, v, t
    else:
        # Return just u and t as in the vib_undamped.py's solver
        return u, t
```


We provide the option that this `solver` function returns the same dat `solver` function from Section 2.1 (if `return_v` is `False`), but we may along with `u` and `t`.

The error in the Taylor series expansion behind (48) is $\mathcal{O}(\Delta t^3)$, w error in the central difference for v is $\mathcal{O}(\Delta t^2)$. The overall error is then n than $\mathcal{O}(\Delta t^2)$, which can be verified empirically using the `convergence` function from 2.2:

```
>>> import vib_undamped_velocity_Verlet as m
>>> m.convergence_rates(4, solver_function=m.solver)
[2.0036366687367346, 2.0009497328124835, 2.000240105995295]
```

9 Generalization: damping, nonlinear sprin external excitation

We shall now generalize the simple model problem from Section 1 to ir possibly nonlinear damping term $f(u')$, a possibly nonlinear spring (or re force $s(u)$, and some external excitation $F(t)$:

$$mu'' + f(u') + s(u) = F(t), \quad u(0) = I,\ u'(0) = V,\ t \in (0,T]\,.$$

We have also included a possibly nonzero initial value of $u'(0)$. The par m, $f(u')$, $s(u)$, $F(t)$, I, V, and T are input data.

There are two main types of damping (friction) forces: linear $f(u') =$ quadratic $f(u') = bu'|u'|$. Spring systems often feature linear dampin air resistance usually gives rise to quadratic damping. Spring forces a linear: $s(u) = cu$, but nonlinear versions are also common, the most fa the gravity force on a pendulum that acts as a spring with $s(u) \sim \sin(u$

9.1 A centered scheme for linear damping

Sampling (50) at a mesh point t_n, replacing $u''(t_n)$ by $[D_t D_t u]^n$, and u $[D_{2t}u]^n$ results in the discretization

$$[mD_t D_t u + f(D_{2t}u) + s(u) = F]^n,$$

which written out means

$$m\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} + f(\frac{u^{n+1} - u^{n-1}}{2\Delta t}) + s(u^n) = F^n,$$

where F^n as usual means $F(t)$ evaluated at $t = t_n$. Solving (52) with re the unknown u^{n+1} gives a problem: the u^{n+1} inside the f function ma equation *nonlinear* unless $f(u')$ is a linear function, $f(u') = bu'$. For shall assume that f is linear in u'. Then

$$m\frac{u^{n+1}-2u^n+u^{n-1}}{\Delta t^2}+b\frac{u^{n+1}-u^{n-1}}{2\Delta t}+s(u^n)=F^n, \tag{53}$$

hich gives an explicit formula for u at each new time level:

$$u^{n+1}=(2mu^n+(\frac{b}{2}\Delta t-m)u^{n-1}+\Delta t^2(F^n-s(u^n)))(m+\frac{b}{2}\Delta t)^{-1}\,. \tag{54}$$

For the first time step we need to discretize $u'(0)=V$ as $[D_{2t}u=V]^0$ and ombine with (54) for $n=0$. The discretized initial condition leads to

$$u^{-1}=u^1-2\Delta tV, \tag{55}$$

hich inserted in (54) for $n=0$ gives an equation that can be solved for u^1:

$$u^1=u^0+\Delta t\,V+\frac{\Delta t^2}{2m}(-bV-s(u^0)+F^0)\,. \tag{56}$$

.2 A centered scheme for quadratic damping

/hen $f(u')=bu'|u'|$, we get a quadratic equation for u^{n+1} in (52). This quation can straightforwardly be solved, but we can also avoid the nonlinearity y performing an approximation that is within other numerical errors that we ave already committed by replacing derivatives with finite differences.

The idea is to reconsider (50) and only replace u'' by D_tD_tu, giving

$$[mD_tD_tu+bu'|u'|+s(u)=F]^n, \tag{57}$$

ere, $u'|u'|$ is to be computed at time t_n. We can introduce a *geometric mean*, efined by

$$(w^2)^n\approx w^{n-\frac{1}{2}}w^{n+\frac{1}{2}},$$

or some quantity w depending on time. The error in the geometric mean pproximation is $\mathcal{O}(\Delta t^2)$, the same as in the approximation $u''\approx D_tD_tu$. With $=u'$ it follows that

$$[u'|u'|]^n\approx u'(t_n+\frac{1}{2})|u'(t_n-\frac{1}{2})|\,.$$

he next step is to approximate u' at $t_{n\pm1/2}$, but here a centered difference can e used:

$$u'(t_{n+1/2})\approx[D_tu]^{n+\frac{1}{2}},\quad u'(t_{n-1/2})\approx[D_tu]^{n-\frac{1}{2}}\,. \tag{58}$$

/e then get

$$[u'|u'|]^n\approx[D_tu]^{n+\frac{1}{2}}|[D_tu]^{n-\frac{1}{2}}|=\frac{u^{n+1}-u^n}{\Delta t}\frac{|u^n-u^{n-1}|}{\Delta t}\,. \tag{59}$$

The counterpart to (52) is then

$$m\frac{u^{n+1}-2u^n+u^{n-1}}{\Delta t^2}+b\frac{u^{n+1}-u^n}{\Delta t}\frac{|u^n-u^{n-1}|}{\Delta t}+s(u^n)=F^n,$$

which is linear in u^{n+1}. Therefore, we can easily solve with respect to u achieve the explicit updating formula

$$u^{n+1}=\left(m+b|u^n-u^{n-1}|\right)^{-1}\times \left(2mu^n-mu^{n-1}+bu^n|u^n-u^{n-1}|+\Delta t^2(F^n-s(u^n))\right)\,.$$

In the derivation of a special equation for the first time step we some trouble: inserting (55) in (61) for $n=0$ results in a complicated n equation for u^1. By thinking differently about the problem we can ea away with the nonlinearity again. We have for $n=0$ that $b[u'|u'|]^0$ = Using this value in (57) gives

$$[mD_tD_tu+bV|V|+s(u)=F]^0\,.$$

Writing this equation out and using (55) results in the special equation first time step:

$$u^1=u^0+\Delta tV+\frac{\Delta t^2}{2m}\left(-bV|V|-s(u^0)+F^0\right)\,.$$

9.3 A forward-backward discretization of the qua damping term

The previous section first proposed to discretize the quadratic dampi $|u'|u'$ using centered differences: $[|D_{2t}|D_{2t}u]^n$. As this gives rise to a nonl in u^{n+1}, it was instead proposed to use a geometric mean combined with differences. But there are other alternatives. To get rid of the nonline $[|D_{2t}|D_{2t}u]^n$, one can think differently: apply a backward difference to $|$ that the term involves known values, and apply a forward difference make the term linear in the unknown u^{n+1}. With mathematics,

$$[\beta|u'|u']^n\approx\beta|[D_t^-u]^n|[D_t^+u]^n=\beta\left|\frac{u^-u^{n-1}}{\Delta t}\right|\frac{u^{n+1}-u^n}{\Delta t}\,.$$

The forward and backward differences have both an error proportional t one may think the discretization above leads to a first-order scheme. How looking at the formulas, we realize that the forward-backward differenc in exactly the same scheme as when we used a geometric mean and differences. Therefore, the forward-backward differences act in a symme and actually produce a second-order accurate discretization of the qu damping term.

.4 Implementation

he algorithm arising from the methods in Sections 9.1 and 9.2 is very similar
› the undamped case in Section 1.2. The difference is basically a question
f different formulas for u^1 and u^{n+1}. This is actually quite remarkable. The
quation (50) is normally impossible to solve by pen and paper, but possible
›r some special choices of F, s, and f. On the contrary, the complexity of the
onlinear generalized model (50) versus the simple undamped model is not a big
eal when we solve the problem numerically!

The computational algorithm takes the form

1. $u^0 = I$
2. compute u^1 from (56) if linear damping or (63) if quadratic damping
3. for $n = 1, 2, \ldots, N_t - 1$:
   (a) compute u^{n+1} from (54) if linear damping or (61) if quadratic damping

Iodifying the `solver` function for the undamped case is fairly easy, the big
ifference being many more terms and if tests on the type of damping:

```
def solver(I, V, m, b, s, F, dt, T, damping='linear'):
    """
    Solve m*u'' + f(u') + s(u) = F(t) for t in (0,T],
    u(0)=I and u'(0)=V,
    by a central finite difference method with time step dt.
    If damping is 'linear', f(u')=b*u, while if damping is
    'quadratic', f(u')=b*u'*abs(u').
    F(t) and s(u) are Python functions.
    """
    dt = float(dt); b = float(b); m = float(m) # avoid integer div.
    Nt = int(round(T/dt))
    u = zeros(Nt+1)
    t = linspace(0, Nt*dt, Nt+1)

    u[0] = I
    if damping == 'linear':
        u[1] = u[0] + dt*V + dt**2/(2*m)*(-b*V - s(u[0]) + F(t[0]))
    elif damping == 'quadratic':
        u[1] = u[0] + dt*V + \
               dt**2/(2*m)*(-b*V*abs(V) - s(u[0]) + F(t[0]))

    for n in range(1, Nt):
        if damping == 'linear':
            u[n+1] = (2*m*u[n] + (b*dt/2 - m)*u[n-1] +
                      dt**2*(F(t[n]) - s(u[n])))/(m + b*dt/2)
        elif damping == 'quadratic':
            u[n+1] = (2*m*u[n] - m*u[n-1] + b*u[n]*abs(u[n] - u[n-1])
                      + dt**2*(F(t[n]) - s(u[n])))/\
                      (m + b*abs(u[n] - u[n-1]))
    return u, t
```

he complete code resides in the file `vib.py`[13].

[13] `http://tinyurl.com/nm5587k/vib/vib.py`

9.5 Verification

Constant solution. For debugging and initial verification, a constant
is often very useful. We choose $u_e(t) = I$, which implies $V = 0$. Inserte
ODE, we get $F(t) = s(I)$ for any choice of f. Since the discrete derivat
constant vanishes (in particular, $[D_{2t}I]^n = 0$, $[D_tI]^n = 0$, and $[D_tD_tI$
the constant solution also fulfills the discrete equations. The constant
therefore be reproduced to machine precision.

Linear solution. Now we choose a linear solution: $u_e = ct + d$. Th
condition $u(0) = I$ implies $d = I$, and $u'(0) = V$ forces c to be V. In
$u_e = Vt + I$ in the ODE with linear damping results in

$$0 + bV + s(Vt + I) = F(t),$$

while quadratic damping requires the source term

$$0 + b|V|V + s(Vt + I) = F(t)\,.$$

Since the finite difference approximations used to compute u' all are exa
linear function, it turns out that the linear u_e is also a solution of the
equations. Exercise 9 asks you to carry out all the details.

Quadratic solution. Choosing $u_e = bt^2 + Vt + I$, with b arbitrary
the initial conditions and fits the ODE if F is adjusted properly. The
also solves the discrete equations with linear damping. However, this qu
polynomial in t does not fulfill the discrete equations in case of quadratic d
because the geometric mean used in the approximation of this term int
an error. Doing Exercise 9 will reveal the details. One can fit F^n in the
equations such that the quadratic polynomial is reproduced by the nu
method (to machine precision).

9.6 Visualization

The functions for visualizations differ significantly from those in the un
case in the `vib_undamped.py` program because we in the present general
not have an exact solution to include in the plots. Moreover, we have
estimate of the periods of the oscillations as there will be one period det
by the system parameters, essentially the approximate frequency $\sqrt$
for linear s and small damping, and one period dictated by $F(t)$ in
excitation is periodic. This is, however, nothing that the program can
on or make use of. Therefore, the user has to specify T and the windo
in case of a plot that moves with the graph and shows the most recent
it in long time simulations.

The `vib.py` code contains several functions for analyzing the tim
signal and for visualizing the solutions.

.7 User interface

he `main` function has substantial changes from the `vib_undamped.py` code nce we need to specify the new data c, $s(u)$, and $F(t)$. In addition, we must et T and the plot window width (instead of the number of periods we want to mulate as in `vib_undamped.py`). To figure out whether we can use one plot or the whole time series or if we should follow the most recent part of u, we an use the `plot_empricial_freq_and_amplitude` function's estimate of the umber of local maxima. This number is now returned from the function and sed in `main` to decide on the visualization technique.

```
def main():
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument('--I', type=float, default=1.0)
    parser.add_argument('--V', type=float, default=0.0)
    parser.add_argument('--m', type=float, default=1.0)
    parser.add_argument('--c', type=float, default=0.0)
    parser.add_argument('--s', type=str, default='u')
    parser.add_argument('--F', type=str, default='0')
    parser.add_argument('--dt', type=float, default=0.05)
    parser.add_argument('--T', type=float, default=140)
    parser.add_argument('--damping', type=str, default='linear')
    parser.add_argument('--window_width', type=float, default=30)
    parser.add_argument('--savefig', action='store_true')
    a = parser.parse_args()
    from scitools.std import StringFunction
    s = StringFunction(a.s, independent_variable='u')
    F = StringFunction(a.F, independent_variable='t')
    I, V, m, c, dt, T, window_width, savefig, damping = \
       a.I, a.V, a.m, a.c, a.dt, a.T, a.window_width, a.savefig, \
       a.damping

    u, t = solver(I, V, m, c, s, F, dt, T)
    num_periods = empirical_freq_and_amplitude(u, t)
    if num_periods <= 15:
        figure()
        visualize(u, t)
    else:
        visualize_front(u, t, window_width, savefig)
    show()
```

he program `vib.py` contains the above code snippets and can solve the model roblem (50). As a demo of `vib.py`, we consider the case $I = 1$, $V = 0$, $m = 1$, $= 0.03$, $s(u) = \sin(u)$, $F(t) = 3\cos(4t)$, $\Delta t = 0.05$, and $T = 140$. The relevant ommand to run is

```
erminal> python vib.py --s 'sin(u)' --F '3*cos(4*t)' --c 0.03
```

his results in a moving window following the function[14] on the screen. Figure 11 ows a part of the time series.

[14] `http://tinyurl.com/opdfafk/pub/mov-vib/vib_generalized_dt0.05/index.html`

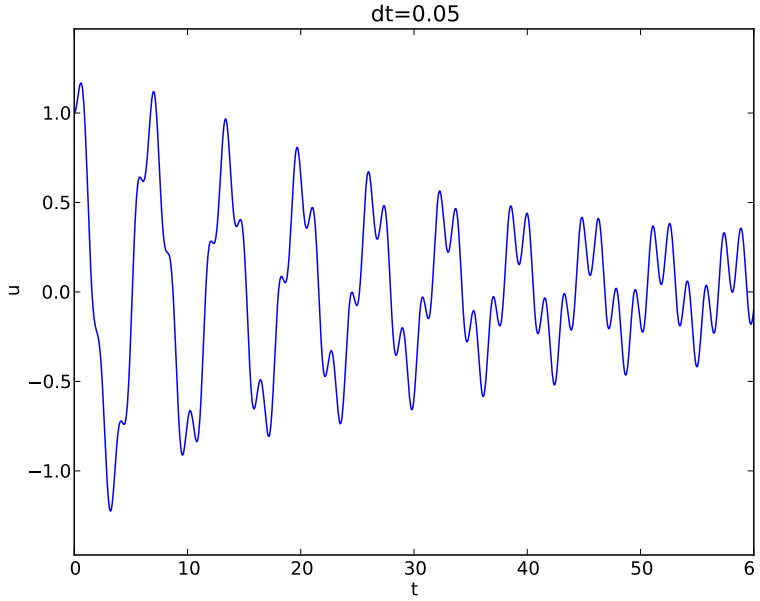


Figure 11: Damped oscillator excited by a sinusoidal function.

9.8 The Euler-Cromer scheme for the generalized n

The ideas of the Euler-Cromer method from Section 8 carry over to the gen model. We write (50) as two equations for u and $v = u'$. The first equ taken as the one with v' on the left-hand side:

$$v' = \frac{1}{m}(F(t) - s(u) - f(v)),$$
$$u' = v\,.$$

The idea is to step (64) forward using a standard Forward Euler metho we update u from (65) with a Backward Euler method, utilizing the computed v^{n+1} value. In detail,

$$\frac{v^{n+1} - v^n}{\Delta t} = \frac{1}{m}(F(t_n) - s(u^n) - f(v^n)),$$
$$\frac{u^{n+1} - u^n}{\Delta t} = v^{n+1},$$

resulting in the explicit scheme

$$v^{n+1} = v^n + \Delta t \frac{1}{m}(F(t_n) - s(u^n) - f(v^n)), \tag{68}$$

$$u^{n+1} = u^n + \Delta t\, v^{n+1}\,. \tag{69}$$

Ve immediately note one very favorable feature of this scheme: all the nonlinarities in $s(u)$ and $f(v)$ are evaluated at a previous time level. This makes the uler-Cromer method easier to apply and hence much more convenient than the entered scheme for the second-order ODE (50).

The initial conditions are trivially set as

$$v^0 = V, \tag{70}$$

$$u^0 = I\,. \tag{71}$$

.0 Exercises and Problems

'roblem 1: Use linear/quadratic functions for verification

onsider the ODE problem

$$u'' + \omega^2 u = f(t), \quad u(0) = I,\ u'(0) = V,\ t \in (0, T]\,.$$

iscretize this equation according to $[D_t D_t u + \omega^2 u = f]^n$.

) Derive the equation for the first time step (u^1).

) For verification purposes, we use the method of manufactured solutions MMS) with the choice of $u_{\rm e}(x,t) = ct + d$. Find restrictions on c and d from ıe initial conditions. Compute the corresponding source term f by term. how that $[D_t D_t t]^n = 0$ and use the fact that the $D_t D_t$ operator is linear, $D_t D_t(ct + d)]^n = c[D_t D_t t]^n + [D_t D_t d]^n = 0$, to show that $u_{\rm e}$ is also a perfect olution of the discrete equations.

) Use `sympy` to do the symbolic calculations above. Here is a sketch of the rogram `vib_undamped_verify_mms.py`:

```
import sympy as sp
V, t, I, w, dt = sp.symbols('V t I w dt')  # global symbols
f = None  # global variable for the source term in the ODE

def ode_source_term(u):
    """Return the terms in the ODE that the source term
    must balance, here u'' + w**2*u.
    u is symbolic Python function of t."""
    return sp.diff(u(t), t, t) + w**2*u(t)

def residual_discrete_eq(u):
    """Return the residual of the discrete eq. with u inserted."""
    R = ...
    return sp.simplify(R)

def residual_discrete_eq_step1(u):
    """Return the residual of the discrete eq. at the first
    step with u inserted."""
    R = ...
    return sp.simplify(R)

def DtDt(u, dt):
    """Return 2nd-order finite difference for u_tt.
    u is a symbolic Python function of t.
    """
    return ...

def main(u):
    """
    Given some chosen solution u (as a function of t, implemented
    as a Python function), use the method of manufactured solutio
    to compute the source term f, and check if u also solves
    the discrete equations.
    """
    print '=== Testing exact solution: %s ===' % u
    print "Initial conditions u(0)=%s, u'(0)=%s:" % \
          (u(t).subs(t, 0), sp.diff(u(t), t).subs(t, 0))

    # Method of manufactured solution requires fitting f
    global f  # source term in the ODE
    f = sp.simplify(ode_lhs(u))

    # Residual in discrete equations (should be 0)
    print 'residual step1:', residual_discrete_eq_step1(u)
    print 'residual:', residual_discrete_eq(u)

def linear():
    main(lambda t: V*t + I)

if __name__ == '__main__':
    linear()
```

Fill in the various functions such that the calls in the `main` function wc

d) The purpose now is to choose a quadratic function $u_{\rm e} = bt^2 + ct + d$ solution. Extend the `sympy` code above with a function `quadratic` for t and checking if the discrete equations are fulfilled. (The function is very to `linear`.)

e) Will a polynomial of degree three fulfill the discrete equations?

f) Implement a `solver` function for computing the numerical solutior problem.

g) Write a nose test for checking that the quadratic solution is comp correctly (too machine precision, but the round-off errors accumulate and with T) by the `solver` function.
Filenames: `vib_undamped_verify_mms.pdf`, `vib_undamped_verify_mı`

Exercise 2: Show linear growth of the phase with time

Consider an exact solution $I\cos(\omega t)$ and an approximation $I\cos(\tilde{\omega}t)$. Define the phase error as time lag between the peak I in the exact solution and the corresponding peak in the approximation after m periods of oscillations. Show that this phase error is linear in m. Filename: `vib_phase_error_growth.pdf`.

Exercise 3: Improve the accuracy by adjusting the frequency

According to (18), the numerical frequency deviates from the exact frequency by (dominating) amount $\omega^3\Delta t^2/24 > 0$. Replace the `w` parameter in the algorithm in the `solver` function in `vib_undamped.py` by `w*(1 - (1./24)*w**2*dt**2` and test how this adjustment in the numerical algorithm improves the accuracy (use $\Delta t = 0.1$ and simulate for 80 periods, with and without adjustment of ω). Filename: `vib_adjust_w.py`.

Exercise 4: See if adaptive methods improve the phase error

Adaptive methods for solving ODEs aim at adjusting Δt such that the error is within a user-prescribed tolerance. Implement the equation $u'' + u = 0$ in the Odespy[15] software. Use the example from Section **??**in [1]. Run the scheme with a very low tolerance (say 10^{-14}) and for a long time, check the number of time points in the solver's mesh (`len(solver.t_all)`), and compare the phase error with that produced by the simple finite difference method from Section 1.2 with the same number of (equally spaced) mesh points. The question is whether it pays off to use an adaptive solver or if equally many points with a simple method gives about the same accuracy. Filename: `vib_undamped_adaptive.py`.

Exercise 5: Use a Taylor polynomial to compute u^1

As an alternative to the derivation of (8) for computing u^1, one can use a Taylor polynomial with three terms for u^1:

$$u(t_1) \approx u(0) + u'(0)\Delta t + \frac{1}{2}u''(0)\Delta t^2$$

With $u'' = -\omega^2 u$ and $u'(0) = 0$, show that this method also leads to (8). Generalize the condition on $u'(0)$ to be $u'(0) = V$ and compute u^1 in this case with both methods. Filename: `vib_first_step.pdf`.

[15] `https://github.com/hplgit/odespy`

Exercise 6: Find the minimal resolution of an oscil function

Sketch the function on a given mesh which has the highest possible fre That is, this oscillatory "cos-like" function has its maxima and mi every two grid points. Find an expression for the frequency of this f and use the result to find the largest relevant value of $\omega\Delta t$ when ω frequency of an oscillating function and Δt is the mesh spacing. Fi `vib_largest_wdt.pdf`.

Exercise 7: Visualize the accuracy of finite differenc a cosine function

We introduce the error fraction

$$E = \frac{[D_t D_t u]^n}{u''(t_n)}$$

to measure the error in the finite difference approximation $D_t D_t u$ to u''. C E for the specific choice of a cosine/sine function of the form $u = \exp(i$ show that

$$E = \left(\frac{2}{\omega\Delta t}\right)^2 \sin^2(\frac{\omega\Delta t}{2}).$$

Plot E as a function of $p = \omega\Delta t$. The relevant values of p are $[0, \pi]$ (see Ex for why $p > \pi$ does not make sense). The deviation of the curve from un alizes the error in the approximation. Also expand E as a Taylor polynor up to fourth degree (use, e.g., `sympy`). Filename: `vib_plot_fd_exp_er`

Exercise 8: Verify convergence rates of the error in e

We consider the ODE problem $u'' + \omega^2 u = 0$, $u(0) = I$, $u'(0) = V$, for t The total energy of the solution $E(t) = \frac{1}{2}(u')^2 + \frac{1}{2}\omega^2 u^2$ should stay c The error in energy can be computed as explained in Section 7.

Make a nose test in a file `test_error_conv.py`, where code from `vib_` is imported, but the `convergence_rates` and `test_convergence_rat` tions are copied and modified to also incorporate computations of the energy and the convergence rate of this error. The expected rate is 2. F `test_error_conv.py`.

Exercise 9: Use linear/quadratic functions for verific

This exercise is a generalization of Problem 1 to the extended model p (50) where the damping term is either linear or quadratic. Solve the subproblems and see how the results and problem settings change w generalized ODE in case of linear or quadratic damping. By modify code from Problem 1, `sympy` will do most of the work required to ana generalized problem. Filename: `vib_verify_mms.py`.

Exercise 10: Use an exact discrete solution for verification

Write a nose test function in a separate file that employs the exact discrete solution (19) to verify the implementation of the `solver` function in the file `vib_undamped.py`. Filename: `test_vib_undamped_exact_discrete_sol.py`.

Exercise 11: Use analytical solution for convergence rate tests

The purpose of this exercise is to perform convergence tests of the problem (50) when $s(u) = \omega^2 u$ and $F(t) = A\sin\phi t$. Find the complete analytical solution to the problem in this case (most textbooks on mechanics or ordinary differential equations list the various elements you need to write down the exact solution). Modify the `convergence_rate` function from the `vib_undamped.py` program to perform experiments with the extended model. Verify that the error is of order Δt^2. Filename: `vib_conv_rate.py`.

Exercise 12: Investigate the amplitude errors of many solvers

Use the program `vib_undamped_odespy.py` from Section 6 and the amplitude estimation from the `amplitudes` function in the `vib_undamped.py` file (see Section 3.4) to investigate how well famous methods for 1st-order ODEs can preserve the amplitude of u in undamped oscillations. Test, for example, the 3rd- and 4th-order Runge-Kutta methods (`RK3`, `RK4`), the Crank-Nicolson method (`CrankNicolson`), the 2nd- and 3rd-order Adams-Bashforth methods (`AdamsBashforth2`, `AdamsBashforth3`), and a 2nd-order Backwards scheme (`Backward2Step`). The relevant governing equations are listed in Section 23. Filename: `vib_amplitude_errors.py`.

Exercise 13: Minimize memory usage of a vibration solver

The program `vib.py`[16] store the complete solution $u^0, u^1, \ldots, u^{N_t}$ in memory, which is convenient for later plotting. Make a memory minimizing version of this program where only the last three u^{n+1}, u^n, and u^{n-1} values are stored in memory. Write each computed (t_{n+1}, u^{n+1}) pair to file. Visualize the data in the file (a cool solution is to read one line at a time and plot the u value using the line-by-line plotter in the `visualize_front_ascii` function - this technique makes it trivial to visualize very long time simulations). Filename: `vib_memsave.py`.

Exercise 14: Implement the solver via classes

Reimplement the `vib.py` program using a class `Problem` to hold all the physical parameters of the problem, a class `Solver` to hold the numerical parameters and compute the solution, and a class `Visualizer` to display the solution.

[16] `http://tinyurl.com/nm5587k/vib/vib.py`

Hint. Use the ideas and examples from Section ?? and ??in [1]. More cally, make a superclass `Problem` for holding the scalar physical paramet problem and let subclasses implement the $s(u)$ and $F(t)$ functions as n Try to call up as much existing functionality in `vib.py` as possible. Filename: `vib_class.py`.

Exercise 15: Interpret $[D_t D_t u]^n$ as a forward-backwa ference

Show that the difference $[D_t D_t u]^n$ is equal to $[D_t^+ D_t^- u]^n$ and $D_t^- D_t^+ u]$ is, instead of applying a centered difference twice one can alternatively mixture forward and backward differences. Filename: `vib_DtDt_fw_bw`

Exercise 16: Use the forward-backward scheme with q damping

We consider the generalized model with quadratic damping, express system of two first-order equations as in Section ??:

$$u' = v,$$
$$v' = \frac{1}{m}\left(F(t) - \beta|v|v - s(u)\right).$$

However, contrary to what is done in Section ??, we want to apply the the forward-backward discretization in Section 8. Express the idea in c notation and write out the scheme. Unfortunately, the backward differ the v equation creates a nonlinearity $|v^{n+1}|v^n$. To linearize this nonlinea the known value v^n inside the absolute value factor, i.e., $|v^{n+1}|v^n \approx |$ Show that the resulting scheme is equivalent to the one in Section ?? f time level $n \geq 1$.

What we learn from this exercise is that the first-order differences linearization trick play together in "the right way" such that the schei good as when we (in Section ??) carefully apply centered difference geometric mean on a staggered mesh to achieve second-order accuracy. a difference in the handling of the initial conditions, though, as explaine end of Section 8. Filename: `vib_gen_bwdamping.pdf`.

Exercise 17: Use a backward difference for the da term

As an alternative to discretizing the damping terms $\beta u'$ and $\beta|u'|u'$ by c differences, we may apply backward differences:

$$[u']^n \approx [D_t^- u]^n,$$
$$[|u'|u']^n \approx [|D_t^- u|D_t^- u]^n = |[D_t^- u]^n|[D_t^- u]^n.$$

he advantage of the backward difference is that the damping term is evalu-
ted using known values u^n and u^{n-1} only. Extend the `vib.py`[17] code with
scheme based on using backward differences in the damping terms. Add
tatements to compare the original approach with centered difference and the
ew idea launched in this exercise. Perform numerical experiments to investigate
ow much accuracy that is lost by using the backward differences. Filename:
`ib_gen_bwdamping.pdf`.

Exercise 18: Simulate a bouncing ball

bouncing ball is a body in free vertically fall until it impacts the ground.
uring the impact, some kinetic energy is lost, and a new motion upwards with
educed velocity starts. At some point the velocity close to the ground is so
nall that the ball is considered to be finally at rest.

The motion of the ball falling in air is governed by Newton's second law
$= ma$, where a is the acceleration of the body, m is the mass, and F is the
m of all forces. Here, we neglect the air resistance so that gravity $-mg$ is the
nly force. The height of the ball is denoted by h and v is the velocity. The
elations between h, v, and a,

$$h'(t) = v(t), \quad v'(t) = a(t),$$

ombined with Newton's second law gives the ODE model

$$h''(t) = -g, \tag{72}$$

r expressed alternatively as a system of first-order equations:

$$v'(t) = -g, \tag{73}$$
$$h'(t) = v(t)\,. \tag{74}$$

hese equations govern the motion as long as the ball is away from the ground
y a small distance $\epsilon_h > 0$. When $h < \epsilon_h$, we have two cases.

1. The ball impacts the ground, recognized by a sufficiently large negative velocity ($v < -\epsilon_v$). The velocity then changes sign and is reduced by a factor C_R, known as the coefficient of restitution[18]. For plotting purposes, one may set $h = 0$.

2. The motion stops, recognized by a sufficiently small velocity ($|v| < \epsilon_v$) close to the ground.

hoose one of the models, (72) or (73)-(74), and simulate a bouncing ball. Plot
(t). Think about how to plot $v(t)$.

[17] `http://tinyurl.com/nm5587k/vib/vib.py`
[18] `http://en.wikipedia.org/wiki/Coefficient_of_restitution`

Hint. A naive implementation may get stuck in repeated impacts f
time step sizes. To avoid this situation, one can introduce a state varia
holds the mode of the motion: free fall, impact, or rest. Two consecutive
imply that the motion has stopped.
Filename: `bouncing_ball.py`.

Exercise 19: Simulate an elastic pendulum

Consider an elastic pendulum fixed to the point $\boldsymbol{r}_0 = (0, L_0)$. The le
the pendulum wire when not stretched is L_0. The wire is massless and
straight. At the end point $\boldsymbol{r}$, we have a mass m. Stretching the elasti
distance s gives rise to a spring force ks in the opposite direction of the st
Let $\boldsymbol{n}$ be a unit normal vector along the wire:

$$\boldsymbol{n} = \frac{\boldsymbol{r} - \boldsymbol{r}_0}{||\boldsymbol{r} - \boldsymbol{r}_0||}\,.$$

The stretch s in the wire is the current length $||\boldsymbol{r} - \boldsymbol{r}_0||$ at some time
the original length L_0:

$$s = ||\boldsymbol{r} - \boldsymbol{r}_0|| - L_0\,.$$

The force in the wire is then $\boldsymbol{F}_w = -ks\boldsymbol{n}$. Newton's second law of motion
to the mass results in

$$m\ddot{\boldsymbol{r}} = -ks\boldsymbol{n} - mg\boldsymbol{j},$$

where $\boldsymbol{j}$ is a unit vector in the upward vertical direction. Let $\boldsymbol{r} = (x$
$\boldsymbol{n} = (n_x, n_y)$. The two components of (75) then becomes

$$\ddot{x} = -m^{-1}ksn_x,$$

$$\ddot{y} = -m^{-1}ksn_y - g\,.$$

The mass point (x, y) will undergo a two-dimensional motion, but if the
stiff (large k), the elastic pendulum will approach the classical one wh
mass point moves along a circle. However, the elastic pendulum is n
by a direct application of Newton's second law, because the force in t
is known (as a function of the motion), while the classical pendulum i
constrained motion, which requires elimination of an unknown via the co
(by invoking polar coordinates).

In equilibrium, the mass hangs in the position $(0, y_0)$ determined by

$$0 = -m^{-1}ksn_y - g = m^{-1}k(y_0 - L_0) - g \quad \Rightarrow \quad y_0 = L_0 + mg/k =$$

We then displace the mass an angle θ_0 to the right. The initial position $(x($
then becomes

$$x(0) = L\sin\theta_0, \quad y(0) = L_0 - L\cos\theta_0\,.$$

he velocity is zero, $x'(0) = y'(0) = 0$.

) Write a code that can simulate such an elastic pendulum. Plot y against x in plot with the same length scale on the axis such that we get a correct picture f the motion. Also plot the angle the pendulum: $\theta = \tan^{-1} x/(L_0 - y)$.

A possible set of parameters is $L_0 = 9.81$ m, $m = 1$ kg, $theta_0 = 30$ degrees.

lint 1. The associated classical pendulum, for large k, has an equation $+ g/L_0\theta = 0$. With $g = L_0$, the period is 2π: $\theta(t) = \theta_0\cos(t)$. One can ompare in a plot the angle of the elastic solution ($\theta = \tan^{-1} x/(L_0 - y)$) with ie solution of the classical pendulum problem.

lint 2. The equation of motion is subject to round-off errors for large k, ecause s is then small such that ks is a product of a large and a small number. Moreover, in this stiff case, $m^{-1}ksn_y$ is close to g such that we also subtract wo almost equal numbers in the force term in the equation. For the given arameters, $k = 150$ is a large value and gives a solution close to the motion of perfect classical pendulum. Much larger values gives unstable solutions.

) Air resistance is a force $\frac{1}{2}\varrho C_D A||bmv||\boldsymbol{v}$, where C_D is a drag coefficient (0.2 or a sphere), ϱ is the density of air (1.2 kg m^{-3}), A is the cross section area $A = \pi R^2$ for a sphere, where R is the radius), and $\boldsymbol{v}$ is the velocity: $\boldsymbol{v} = \dot{\boldsymbol{r}}$. nclude air resistance in the model and show plots comparing the motion with nd without air resistance.
ilename: `elastic_pendulum.py`.

Exercise 20: Analysis of the Euler-Cromer scheme

he Euler-Cromer scheme for the model problem $u'' + \omega^2 u = 0$, $u(0) = I$, $'(0) = 0$, is given in (45)-(44). Find the exact discrete solutions of this scheme nd show that the solution for u^n coincides with that found in Section 4.

lint. Use an "ansatz" $u^n = I\exp(i\tilde{\omega}\Delta t\, n)$ and $v^{=}qu^n$, where $\tilde{\omega}$ and q are nknown parameters. The formula

$$\exp(i\tilde{\omega}(\Delta t)) + \exp(i\tilde{\omega}(-\Delta t)) - 2 = 2\left(\cosh(i\tilde{\omega}\Delta t) - 1\right) = -4\sin^2(\frac{\tilde{\omega}\Delta t}{2}),$$

ecomes handy.

References

.] H. P. Langtangen. Introduction to computing with finite difference methods. Web document, Simula Research Laboratory and University of Oslo, 2013.

Index